# Practical Scientific Computing in Python

John D. Hunter
Fernando Pérez

with contributions from

Perry Greenfield
Travis E. Oliphant
Prabhu Ramachandran

# Contents

CHAPTER 1

# Python for Scientific Computing

With material contributed by Perry Greenfield, Robert Jedrzejewski, Vicki Laidler and John Hunter

## 1.1. Who is using Python?

The use of Python in scientific computing is as wide as the field itself. A sampling of current work is provided here to indicate the breadth of disciplines represented and the scale of the problems addressed. The NASA Jet Propulsion Laboratory (JPL) uses Python as an interface language to FORTRAN and C++ libraries which form a suite of tools for plotting and visualization of spacecraft trajectory parameters in mission design and navigation. The Space Telescope Science Institute (STScI) uses Python in many phases of their pipeline: scheduling Hubble data acquisitions, managing volumes of data, and analyzing astronomical images [**7**]. The National Oceanic Atmospheric Administration (NOAA) uses Python for a wide variety of scientific computing tasks including simple scripts to parse and translate data files, prototyping of computational algorithms, writing user interfaces, web front ends, and the development of models [**27, 6, 29**]. At the Fundamental Symmetries Lab at Princeton University, Python is used to efficiently analyze large data sets from an experiment that searches for CPT and Lorentz Violation using an atomic magnetometer [**23, 22**]. The Pediatric Clinical Electrophysiology unit at The University of Chicago, which collects approximately 100 GB of data per week, uses Python to explore novel approaches to the localization and detection of epileptic seizures [**19**]. The Enthought Corporation is using Python to build customized applications for oil exploration for the petroleum industry. At the world's largest radio telescopes, e.g., Arecibo and the Green Bank Telescope, Python is used for data processing, modelling, and scripting high-performance computing jobs in order to search for and monitor binary and millisecond pulsars in terabyte datasets [**33, 32**]. At the Computational Genomics Laboratory at the Australian National University, researchers are using Python to build a toolkit which enables the specification of novel statistical models of sequence evolution on parallel hardware [**20, 12**]. Michel Sanner's group at the Scripps Research Institute uses Python extensively to build a suite of applications for molecular visualization and exploration of drug/molecule interactions using virtual reality and 3D printing technology[**36, 37**]. Engineers at Google use Python in automation, control and tuning of their computational grid, and use `SWIG` generated Python of their in-house C++ libraries in virtually all facets of their work [**9, 39**]. Many other use cases – ranging from animation at Industrial Light and Magic, to space shuttle mission control, to grid monitoring and control at Rackspace, to drug discovery, meteorology and air traffic control – are detailed in O'Reilly's two volumes of *Python Success Stories* [**1, 2**].

## 1.2. Advantages of Python

> *The canonical, "Python is a great first language", elicited, "Python is a great last language!"* – Noah Spurrier

This quotation summarizes an important reason scientists migrate to Python as a programming language. As a "great first language" Python has a simple, expressive syntax that is accessible to the newcomer. "Python as executable pseudocode" reflects the fact that Python syntax mirrors the obvious and intuitive pseudo-code syntax used in many journals [**40**]. As a great first language, it does not impose a single programming paradigm on scientists, as Java does with object oriented programming, but rather allows one to code at many levels of sophistication, including BASIC/FORTRAN/Matlab style procedural programming familiar to many scientists. Here is the canonical first program "hello world" in Python:

```
# Python
print 'hello world'
```

in Java

```
// java
class myfirstjavaprog
{
  public static void main(String args[])
  {
    System.out.println("Hello World!");
  }
}
```

In addition to being accessible to new programmers and scientists, Python is powerful enough to manage the complexity of large applications, supporting functional programming, object orienting programming, generic programming and metaprogramming. That Python supports these paradigms suggests why it is also a "great last language": as one increases their programming sophistication, the language scales naturally. By contrast, commercial languages like Matlab and IDL, which also support a simple syntax for simple programs do not scale well to complex programming tasks.

The built-in Python data-types and standard library provide a powerful platform in every distribution [**35, ?**]. The standard data types encompass regular and arbitrary length integers, complex numbers, floating point numbers, strings, lists, associative arrays, sets and more. In the standard library included with every Python distribution are modules for regular expressions, data encodings, multimedia formats, math, networking protocols, binary arrays and files, and much more. Thus one can open a file on a remote web server and work with it as easily as with a local file

```
# this 3 line script downloads and prints the yahoo web site
from urllib import urlopen
for line in urlopen('http://yahoo.com').readlines():
   print line
```

Complementing these built-in features, Python is also readily extensible, giving it a wealth of libraries for scientific computing that have been in development for many years [**13, 14**]. `Numeric Python` supports large array manipulations, math, optimized linear algebra, efficient Fourier transforms and random numbers. `scipy` is a collection of Python wrappers of high performance FORTRAN code (eg LAPACK, ODEPACK) for numerical analysis [**3**]. `IPython` is a command shell ala Mathematica, Matlab and IDL for interactive programming, data exploration and visualization with support for command history, completion, debugging and more. `Matplotlib` is a 2D graphics package for making publication quality graphics with a Matlab compatible syntax that is also embeddable in applications. `f2py`, `SWIG`, `weave`, and `pyrex` are tools for rapidly building Python interfaces to high performance compiled code, `MayaVi` is a user friendly graphical user interface for 3D visualizations built on top of the state-of-the-art Visualization Toolkit [**38**]. `pympi`, `pypar`, `pyro`, `scipy.cow`, and `pyxg` are tools for cluster building and doing parallel, remote and distributed computations. This is a sampling of general purpose libraries for scientific computing in Python, and does not begin to address the many high quality, domain specific libraries that are also available.

All of the infrastructure described above is open source software that is freely distributable for academic and commercial use. In both the educational and scientific arenas, this is a critical point. For education, this platform provides students with tools that they can take with them outside the classroom to their homes and jobs and careers beyond. By contrast, the use commercial tools such as Matlab and IDL limits access to major institutions. For scientists, the use of open source tools is consistent with the scientific principle that all of the steps in an analysis or simulation should be open for review, and with the principle of reproducible research [**11**].

### 1.3. Mixed Language Programming

The programming languages of each generation evolve in part to fix the problems of those that came before [**10**]. FORTRAN, the original high level language of scientific computing [**34**], was designed to allow scientists to express code at a level closer to the language of the problem domain. ALGOL and its successor Pascal, widely used in education in the 1970s, were designed to alleviate some of the perceived problems with FORTRAN and to create a language with a simpler and more expressive syntax [**5, 26**]. Object oriented programming languages evolved to allow a closer correspondence between the code and the physical system it models [**16**], and C++ provided a relatively high performance object orientated implementation compatible with the popular C programming language [**42, 41**]. But implementing object orientation efficiently requires programmers stay

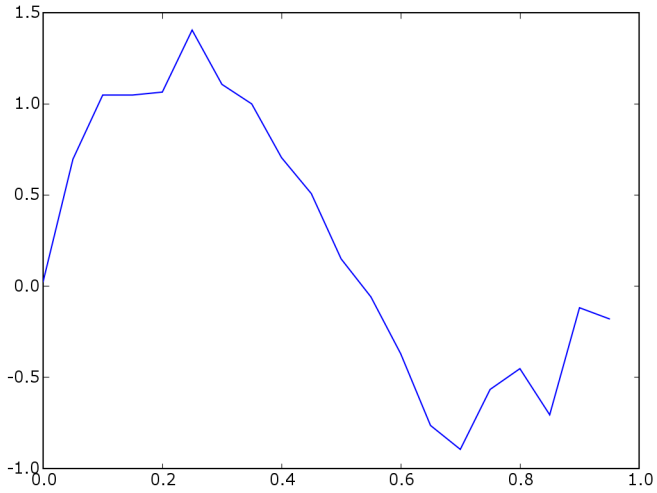



FIGURE 1.4.1. Loading `ASCII` data and displaying with `plot`

close to the machine, managing memory and pointers, and this created a lot of complexity in programs while limiting portability. Interpreted languages such as Tcl, Perl, Python, and Java evolved to manage some of the low-level and platform specific details, making programs easier to write and maintain, but with a performance penalty [**28, 4**]. For many scientists, however, pure object oriented systems like Java are unfamiliar, and languages like Matlab and Python provide the safety, portability and ease of use of an interpreted language without imposing an object oriented approach to coding [**15, 17**].

The result of these several decades is that there are many platforms for scientific computing in use today. The number of man hours invested in numerical methods in FORTRAN, visualization libraries in C++, bioinformatics toolkits in Perl, object frameworks in Java, domain specific toolkits in Matlab, etc... requires an approach that integrates this work. Python is the language that provides maximal integration with other languages, with tools for transparently and semi-automatically interfacing with FORTRAN, C, C++, Java, .NET, Matlab, and Mathematica code [**18, 9**]. In our view, the ability to work seamlessly with code from many languages is the present and the future of scientific computing, and Python effectively integrates these languages into a single environment.

## 1.4. Getting started

We'll get started with python by introducing arrays and plotting by working with a simple ASCII text file `mydata.dat` of two columns; the first column contains the times that some measurement was acquired, and the second column are the sampled voltages at that time. The file looks like

```
0.0000 0.4911
0.0500 0.5012
0.1000 0.7236
0.1500 1.1756
... and so on
```

While it would be easy enough to process this file by writing a python function to do it, there is no need to, since the matplotlib pylab module has a matlab-compatible `load` function for loading ASCII array data (Figure 1.4.1). To complete these exercises, you should have ipython and matplotlib installed, and start ipython in pylab mode with

```
> ipython -pylab
```

LISTING 1.1

```
In [1]: X = load('data/ascii_data.dat') # X is an array
In [2]: t = X[:,0]  # extract the first column
In [3]: v = X[:,1]  # extract the second column
```

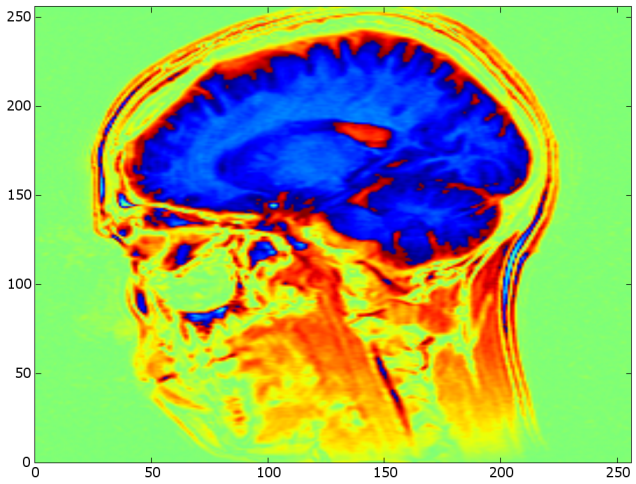



FIGURE 1.4.2. Loading binary image data and displaying with `imshow`

```
In [4]: len(t)
Out[4]: 20
In [5]: len(v)
Out[5]: 20
In [6]: plot(t,v) # plot the data
Out[6]: [<matplotlib.lines.Line2D instance at 0xb65921ac>]
```

It is also easy to load data from binary files. In the example below, we have some image data in raw binary string format. The image is 256x256 pixels, and each pixel is a 2 byte integer. We read this into a string using python's `file` function – the 'rb' flag says to open the file in `read/binary` mode. We can then use the numerix `fromstring` method to convert this to an array, passing the type of the data (`Int16`) as an argument. We reshape the array by changing the array shape attribute to 256 by 256, and pass this off to the matplotlib pylab command `imshow` for plotting. matplotlib has a number of colormaps, and the default one is jet; the data are automatically normalized and colormaps producing the image in Figure 1.4.2

LISTING 1.2

```
# open a file as "read binary" and read it into a string
In [1]: s = file('data/images/r1025.ima', 'rb').read()
# the string is length 256*256*2 = 131072
In [2]: len(s)
Out[2]: 131072
# the data are 2 byte / 16 bit integers
# fromstring converts them to array
In [3]: im = nx.fromstring(s, nx.Int16)
# reshape the array to 256x256
In [4]: im.shape = 256,256
# and plot it with matplotlib's imshow function
In [5]: imshow(im)
Out[5]: <matplotlib.image.AxesImage instance at 0xb659230c>
```

### 1.5. An Introduction to Arrays

**1.5.1. Creating arrays.** There are a few different ways to create arrays besides modules that obtain arrays from data files such

```
>>> x = zeros((20,30))
```

creates a 20x30 array of zeros (default integer type; details on how to specify other types will follow). Note that the dimensions ("shape" in numarray parlance) are specified by giving the dimensions as a comma-separated list within parentheses. The parentheses aren't necessary for a single dimension. As an aside, the parentheses used this way are being used to specify a Python tuple; more will be said about those in a later tutorial. For now you only need to imitate this usage.

Likewise one can create an array of 1's using the `ones()` function.

The `arange()` function can be used to create arrays with sequential values. E.g.,

```
>>> arange(10)
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Note that that the array defaults to starting with a 0 value and does not include the value specified (though the array does have a length that corresponds to the argument)

Other variants:

```
>>> arange(10.)
array([ 0., 1., 2., 3., 4., 5., 6., 7., 8., 9])
>>> arange(3,10)
array([3, 4, 5, 6, 7, 8, 9])
>>> arange(1., 10., 1.1) # note trickiness
array([1. , 2.1, 3.2, 4.3, 5.4, 6.5, 7.6, 8.7, 9.8])
```

Finally, one can create arrays from literal arguments:

```
>>> print array([3,1,7])
[3 1 7]
>>> print array([[2,3],[4,4]])
[[2 3]
 [4 4]]
```

The brackets, like the parentheses in the zeros example above have a special meaning in Python which will be covered later (Python lists). For now, just mimic the syntax used here.

**1.5.2. Array numeric types.** numarray supports all standard numeric types. The default integer matches what Python uses for integers, usually 32 bit integers or what numarray calls `Int32`. The same is true for floats, i.e., generally 64-bit doubles called `Float64` in numarray. The default complex type is `Complex64`. Many of the functions accept a type argument. For example

```
>>> zeros(3, Int8) # Signed byte
>>> zeros(3, type=UInt8) # Unsigned byte
>>> array([2,3], type=Float32)
>>> arange(4, type=Complex64)
```

The possible types are `Int8, UInt8, Int16, UInt16, Int32, UInt32, Int64, UInt64, Float32, Float64, Complex32, Complex64.` To find out the type of an array use the .type() method. E.g.,

```
>>> arr.type()
Float32
```

To convert an array to a different type use the `astype()` method, e.g,

```
>>> a = arr.astype(Float64)
```

**1.5.3. Printing arrays.** Interactively, there are two common ways to see the value of an array. Like many Python objects, just typing the name of the variable itself will print its contents (this only works in interactive mode). You can also explicitly print it. The following illustrates both approaches:

```
>>> x = arange(10)
>>> x
array([0, 1, 2, 3, 4, 5, 6, 7, 8 9])
>>> print x
[0 1 2 3 4 5 6 7 8 9]
```

By default the array module limits the amount of an array that is printed out (to spare you the effects of printing out millions of values). For example:

```
>>> x = arange(1000000)
print x
[    0     1     2 ..., 999997 999998 999999]
```

**1.5.4. Indexing 1-D arrays.** As with IDL and Matlab, there are many options for indexing arrays.

```
>>> x = arange(10)
>>> x
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Simple indexing:

```
>>> x[2] # 3rd element
2
```

Indexing is 0-based. The first value in the array is `x[0]`

Indexing from end:

```
>>> x[-2] # -1 represents the last element, -2 next to last...
8
```

Slices

To select a subset of an array:

```
>>> x[2:5]
array([2, 3, 4])
```

Note that the upper limit of the slice is not included as part of the subset! This is viewed as unexpected by newcomers and a defect. Most find this behavior very useful after getting used to it (the reasons won't be given here). Also important to understand is that slices are views into the original array in the same sense that references view the same array. The following demonstrates:

```
>>> y = x[2:5]
>>> y[0] = 99
>>> y
array([99, 3, 4])
>>> x
array([0, 1, 99, 3, 4, 5, 6, 7, 8, 9])
```

Changes to a slice will show up in the original. If a copy is needed use `x[2:5].copy()`

Short hand notation

```
>>> x[:5] # presumes start from beginning
array([ 0, 1, 99, 3, 4])
>>> x[2:] # presumes goes until end
array([99, 3, 4, 5, 6, 7, 8, 9])
>>> x[:] # selects whole dimension
array([0, 1, 99, 3, 4, 5, 6, 7, 8, 9])
```

Strides:

```
>>> x[2:8:3] # Stride every third element
array([99, 5])
```

Index arrays:

```
>>> x[[4,2,4,1]]
array([4, 99, 4, 1])
```

Using results of logical indexing

```
>>> x > 5
array([0,0,1,0,0,0,1,1,1,1], type=Bool)
>>> x[x>5]
array([99, 6, 7, 8, 9])
```

**1.5.5. Indexing multidimensional arrays.** Before describing this in detail it is very important to note an item regarding multidimensional indexing that will certainly cause you grief until you become accustomed to it: ARRAY INDICES USE THE OPPOSITE CONVENTION AS FORTRAN REGARDING ORDER OF INDICES FOR MULTIDIMENSIONAL ARRAYS.

```
>>> im = arange(24)
>>> im.shape = 4,6
>>> im
array([[ 0,  1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10, 11],
       [12, 13, 14, 15, 16, 17],
       [18, 19, 20, 21, 22, 23]])
```

To emphasize the point made in the previous paragraph, the index that represents the most rapidly varying dimension in memory is the 2nd index, not the first.

Partial indexing:

```
>>> im[1]
array([6, 7, 8, 9, 10, 11])
```

If only some of the indices for a multidimensional array are specified, then the result is an array with the shape of the "leftover" dimensions, in this case, 1-dimensional. The 2nd row is selected, and since there is no index for the column, the whole row is selected.

All of the indexing tools available for 1-D arrays apply to $n$-dimensional arrays as well (though combining index arrays with slices is not currently permitted). To understand all the indexing options in their full detail, read sections 4.6, 4.7 and 6 of the numarray manual.

**1.5.6. Compatibility of dimensions.** In operations involving combining (e.g., adding) arrays or assigning them there are rules regarding the compatibility of the dimensions involved. For example the following is permitted:

```
>>> x[:5] = 0
```

since a single value is considered "broadcastable" over a 5 element array. But this is not permitted:

```
>>> x[:5] = array([0,1,2,3])
```

since a 4 element array does not match a 5 element array.

*The following explanation can probably be skipped by most on the first reading;* it is only important to know that rules for combining arrays of different shapes are quite general. It is hard to precisely specify the rules without getting a bit confusing, but it doesn't take long to get a good intuitive feeling for what is and isn't permitted. Here's an attempt anyway: The shapes of the two involved arrays when aligned on their trailing part must be equal in value or one must have the value one for that dimension. The following pairs of shapes are compatible:

```
(5,4):(4,) -> (5,4)
(5,1):(4,) -> (5,4)
(15,3,5):(15,1,5) -> (15,3,5)
(15,3,5):(3,5) -> (15,3,5)
(15,1,5):(3,1) -> (15,3,5)
```

so that one can add arrays of these shapes or assign one to the other (in which case the one being assigned must be the smaller shape of the two). For the dimensions that have a 1 value that are matched against a larger number, the values in that dimension are simply repeated. For dimensions that are missing, the sub-array is simply repeated for those. The following shapes are not compatible:

```
(3,4):(4,3)
(1,3):(4,)
```

Examples:

```
>>> x = zeros((5,4))
>>> x[:,:] = [2,3,2,3]
>>> x
array([[2, 3, 2, 3],
       [2, 3, 2, 3],
       [2, 3, 2, 3],
       [2, 3, 2, 3],
       [2, 3, 2, 3]])
>>> a = arange(3)
>>> b = a[:] # different array, same data (huh?)
>>> b.shape = (3,1)
>>> b
array([[0],
       [1],
       [2]])
>>> a*b # outer product
array([[0, 0, 0],
       [0, 1, 2],
       [0, 2, 4]])
```

**1.5.7. ufuncs.** A ufunc (short for Universal Function) applies the same operation or function to all the elements of an array independently. When two arrays are added together, the `add` ufunc is used to perform the array addition. There are ufuncs for all the common operations and mathematical functions. More specialized ufuncs can be obtained from add-on libraries. All the operators have corresponding ufuncs that can be used by name (e.g., `add` for `+`). These are all listed in table below. Ufuncs also have a few very handy methods for binary operators and functions whose use are demonstrated here.

```
>>> x = arange(9)
>>> x.shape = (3,3)
>>> x
array([0, 1, 2],
      [3, 4, 5],
      [6, 7, 8]])
>>> add.reduce(x) # sums along the first index
array([9, 12, 15])
>>> add.reduce(x, axis=1) # sums along the 2nd index
array([3, 12, 21])
>>> add.accumulate(x) # cumulative sum along the first index
array([[0,  1,  2],
       [3,  5,  7],
       [9, 12, 15]])
>>> multiply.outer(arange(3),arange(3))
array([[0, 0, 0],
       [0, 1, 2],
       [0, 2, 4]])
```

Standard Ufuncs (with corresponding symbolic operators, when they exist, shown in parentheses)

| | | |
|---|---|---|
| add (+) | log | greater (>) |
| subtract (-) | log10 | greater_equal (>=) |
| multiply (*) | cos | less (<) |
| divide (/) | arcos | less_equal (<=) |
| remainder (%) | sin | logical_and |
| absolute, abs | arcsin | logical_or |
| floor | tan | logical_xor |
| ceil | arctan | bitwise_and (&) |
| fmod | cosh | bitwise_or (\|) |
| conjugate | sinh | bitwise_xor (^) |
| minimum | tanh | bitwise_not (~) |
| maximum | sqrt | rshift (>>) |
| power (**) | equal (==) | lshift (<<) |
| exp | not_equal (!=) | |

*Note that there are no corresponding Python operators for* `logical_and` *and* `logical_or`*. The Python* `and` *and* `or` *operators are NOT equivalent to these respective ufuncs!*

**1.5.8. Array functions.** There are many array utility functions. The following lists the more useful ones with a one line description. See the numarray manual for details on how they are used. Arguments shown with argument=value indicate what the default value is if called without a value for that argument.

`all`(*a*)**::** are all elements of array nonzero
`allclose`(*a1, a2, rtol=1.e-5, atol=1.e-8*)**::** true if all elements within specified amount (between two arrays)
`alltrue`(*a, axis=0*)**::** are all elements nonzero along specified axis true.

`any`(*a*)**::** are any elements of an array nonzero
`argmax`(*a, axis=-1*), argmin(*a,axis=-1*)**::** return array with min/max locations for selected axis
`argsort`(*a, axis=-1*)**::** returns indices of results of sort on an array
`choose`(*selector, population, clipmode=CLIP*)**::** fills specified array by selecting corresponding values from a set of arrays using integer selection array (population is a tuple of arrays; see tutorial 2)
`clip`(*a, amin, amax*)**::** clip values of array *a* at values *amin*, *amax*
`dot`(*a1, a2*)**::** dot product of arrays *a1* & *a2*
`compress`(*condition, a ,axis=0*)**::** selects elements from array *a* based on boolean arraycondition
`concatenate`(*arrays, axis=0*)**::** concatenate arrays contained in sequence of arrays arrays

`cumproduct`(*a, axis=0*):: net cumulative product along specified axis
`cumsum`(*a, axis=0*):: accumulate array along specified axis
`diagonal`(*a, offset=0, axis1=0, axis2=1*):: returns diagonal of 2-d matrix with optional offsets.

`fromfile`(*file, type, shape=None*):: Use binary data in file to form new array of specified type.
`fromstring`(*datastring, type, shape=None*):: Use binary data in *datastring* to form new array of specified shape and type
`identity`(*n, type=None*):: returns identity matrix of size nxn.
`indices`(*shape, type=None*):: generate array with values corresponding to position of selected index of the array
`innerproduct`(*a1, a2*):: guess
`matrixmultiply`(*a1, a2*):: guess
`outerproduct`(*a1, a2*):: guess
`product`(*a, axis=0*):: net product of elements along specified axis
`ravel`(*a*):: creates a 1-d version of an array
`repeat`(*a, repeats, axis=0*):: generates new array with repeated copies of input array *a*
`resize`(*a, shape*):: replicate or truncate array to new shape
`searchsorted`(*bin, a*):: return indices of mapping values of an array *a* into a monotonic array *bin*
`sometrue`(*a, axis=0*):: are any elements along specified axis true
`sort`(*a, axis=-1*):: sort array elements along selected axis
`sum`(*a, axis=0*):: sum array along specified axis
`swapaxes`(*a, axis1, axis2*):: switch indices for axis of array (doesn't actually move data, just maps indices differently)
`trace`(*a, offset=0, axis1=0, axis2=1*):: compute trace of matrix *a* with optional offset.
`transpose`(*a, axes=None*):: transpose indices of array (doesn't actually move data, just maps indices differently)

`where`(*a*):: find "true" locations in array *a*

**1.5.9. Array methods.** Arrays have several methods. They are used as methods are with any object. For example (using the array from the previous example):

```
>>> # sum all array elements
>>> x.sum() # the L indicates a Python Long integer
36L
```

The following lists all the array methods that exist for an array object `a` (a number are equivalent to array functions; these have no summary description shown):

`a.argmax`(*axis=-1*):
`a.argmin`(*axis=-1*):
`a.argsort`(*axis=-1*):
`a.astype`(*type*):: copy array to specified numeric type
`a.byteswap`():: perform byteswap on data in place
`a.byteswapped`():: return byteswapped copy of array
`a.conjugate`():: complex conjugate
`a.copy`():: produce copied version of array (instead of view)
`a.diagonal`():
`a.info`():: print info about array
`a.isaligned`():: are data elements guaranteed aligned with memory?
`a.isbyteswapped`():: are data elements in native processor order?
`a.iscontiguous`():: are data elements contiguous in memory?
`a.is_c_array`():: are data elements aligned, not byteswapped, and contiguous?
`a.is_fortran_contiguous`():: are indicies defined to follow Fortran conventions?
`a.is_f_array`():: are indices defined to follow Fortran conventions and data are aligned and not byteswapped
`a.itemsize`():: size of data element in bytes
`a.max`(type=None):: maximum value in array
`a.min`():: minimum value in array
`a.nelements`():: total number of elements in array
`a.new`():: returns new array of same type and size (data uninitialized)
`a.repeat`(a,repeats,axis=0)::
`a.resize`(shape)::
`a.size`():: same as nelements
`a.type`():: returns type of array
`a.typecode`():: returns corresponding typecode character used by Numeric
`a.tofile`(*file*):: write binary data to file
`a.tolist`():: convert data to Python list format

`a.tostring`():: copy binary data to Python string
`a.transpose`(*axes=-1*):: transpose array
`a.stddev`():: standard deviation
`a.sum`():: sum of all elements
`a.swapaxes`(*axis1,axis2*):
`a.togglebyteorder`():: change byteorder flag without changing actual data byteorder
`a.trace`():
`a.view`():: returns new array object using view of same data

**1.5.10. Array attributes:**
`a.shape`:: returns shape of array
`a.flat`:: returns view of array treating it as 1-dimensional. Doesn't work if array is not contiguous
`a.real`:: return real component of array (exists for all types)
`a.imag, a.imaginary`:: return imaginary component (exists only for complex types)

## 1.6. Exercises

EXERCISE 1.7. Load the binary image shown in Figure1.4.2. What is the mean pixel value, what are the standard deviation of pixel values? Sum over the rows and make a bar plot for the summated intensity across rows. Do the same for columns. Make a histogram of all the data in the image. (Hint – see `nx.mlab.mean`, `nx.mlab.std`, `pylab.bar` and `pylab.hist`)

EXAMPLE 1.7.1. this is another test

this is a test

CHAPTER 2

# A whirlwind tour of python and the standard library

This is a quick-and-dirty introduction to the python language for the impatient scientist. There are many top notch, comprehensive introductions and tutorials for python. For absolute beginners, there is the *Python Beginner's Guide*.[1] The official *Python Tutorial* can be read online[2] or downloaded[3] in a variety of formats. There are over 100 python tutorials collected online.[4]

There are also many excellent books. Targetting newbies is Mark Pilgrim's *Dive into Python* which in available in print and for free online[5], though for absolute newbies even this may be too hard [**31**]. For experienced programmers, David Beasley's *Python Essential Reference* is an excellent introduction to python, but is a bit dated since it only covers python2.1 [**8**]. Likwise Alex Martelli's *Python in a Nutshell* is highly regarded and a bit more current – a 2nd edition is in the works[**24**]. And *The Python Cookbook* is an extremely useful collection of python idioms, tips and tricks [**25**].

But the typical scientist I encounter wants to solve a specific problem, eg, to make a certain kind of graph, to numerically integrate an equation, or to fit some data to a parametric model, and doesn't have the time or interest to read several books or tutorials to get what they want. This guide is for them: a short overview of the language to help them get to what they want as quickly as possible. We get to advanced material pretty quickly, so it may be touch sledding if you are a python newbie. Take in what you can, and if you start getting dizzy, skip ahead to the next section; you can always come back to absorb more detail later, after you get your real work done.

## 2.1. Hello Python

Python is a dynamically typed, object oriented, interpreted language. Interpreted means that your program interacts with the python interpreter, similar to Matlab, Perl, Tcl and Java, and unlike FORTRAN, C, or C++ which are compiled. So let's fire up the python interpreter and get started. I'm not going to cover installing python – it's standard on most linux boxes and for windows there is a friendly GUI installer. To run the python interpreter, on windows, you can click `Start->All Programs->Python 2.4->Python (command line)` or better yet, install `ipython`, a python shell on steroids, and use that. On linux / unix systems, you just need to type `python` or `ipython` at the command line. The `>>>` is the default python shell prompt, so don't type it in the examples below

```
>>> print 'hello world'
hello world
```

As this example shows, *hello world* in python is pretty easy – one common phrase you hear in the python community is that "it fits your brain". – the basic idea is that coding in python feels natural. Compare python's version with *hello world* in C++

```
// C++
#include <iostream>
int main ()
{
  std::cout << "Hello World" << std::endl;
  return 0;
}
```

[1] http://www.python.org/moin/BeginnersGuide
[2] http://docs.python.org/tut/tut.html
[3] http://docs.python.org/download.html
[4] http://www.awaretek.com/tutorials.html
[5] http://diveintopython.org/toc/index.html

### 2.2. Python is a calculator

Aside from my daughter's solar powered cash-register calculator, Python is the only calculator I use. From the python shell, you can type arbitrary arithmetic expressions.

```
>>> 2+2
4
>>> 2**10
1024
>>> 10/5
2
>>> 2+(24.3 + .9)/.24
107.0
>>> 2/3
0
```

The last line is a standard newbie gotcha – if both the left and right operands are integers, python returns an integer. To do floating point division, make sure at least one of the numbers is a float

```
>>> 2.0/3
0.66666666666666663
```

The distinction between integer and floating point division is a common source of frustration among newbies and is slated for destruction in the mythical Python 3000.[6] Since default integer division will be removed in the future, you can invoke the time machine with the `from __future__` directives; these directives allow python programmers today to use features that will become standard in future releases but are not included by default because they would break existing code. From future directives should be among the first lines you type in your python code if you are going to use them, otherwise they may not work. The future division operator will assume floating point division by default,[7]and provides another operator // to do classic integer division.

```
>>> from __future__ import division
>>> 2/3
0.66666666666666663
>>> 2//3
0
```

python has four basic numeric types: int, long, float and complex, but unlike C++, BASIC, FORTRAN or Java, you don't have to declare these types. python can infer them

```
>>> type(1)
<type 'int'>
>>> type(1.0)
<type 'float'>
>>> type(2**200)
<type 'long'>
```

$2^{200}$is a huge number!

```
>>> 2**200
1606938044258990275541962092341162602522202993782792835301376L
```

but python will blithely compute it and much larger numbers for you as long as you have CPU and memory to handle them. The integer type, if it overflows, will automatically convert to a python `long` (as indicated by the appended `L` in the output above) and has no built-in upper bound on size, unlike C/C++ longs.

Python has built in support for complex numbers. Eg, we can verify $i^2 = -1$

```
>>> x = complex(0,1)
>>> x*x
(-1+0j)
```

To access the real and imaginary parts of a complex number, use the `real` and `imag` attributes

---

[6]Python 3000 is a future python release that will clean up several things that Guido considers to be warts.

[7]You may have noticed that 2/3 was represented as 0.66666666666666663 and not 0.66666666666666666 as might be expected. This is because computers are binary calculators, and there is no exact binary representation of 2/3, just as there is no exact binary representation of 0.1

```
>>> 0.1
0.10000000000000001
```

Some languages try and hide this from you, but python is explicit.

```
>>> x.real
0.0
>>> x.imag
1.0
```

If you come from other languages like Matlab, the above may be new to you. In matlab, you might do something like this (>> is the standard matlab shell prompt)

```
>> x = 0+j
x =
   0.0000 + 1.0000i
>> real(x)
ans =
     0
>> imag(x)
ans =
     1
```

That is, in Matlab, you use a *function* to access the real and imaginary parts of the data, but in python these are attributes of the complex object itself. This is a core feature of python and other object oriented languages: an object carries its data and methods around with it. One might say: "a complex number knows it's real and imaginary parts" or "a complex number knows how to take its conjugate", you don't need external functions for these operations

```
>>> x.conjugate
<built-in method conjugate of complex object at 0xb6a62368>
>>> x.conjugate()
-1j
```

On the first line, I just followed along from the example above with `real` and `imag` and typed `x.conjugate` and python printed the representation `<built-in method conjugate of complex object at 0xb6a62368>`. This means that `conjugate` is a *method*, a.k.a a function, and in python we need to use parentheses to call a function. If the method has arguments, like the `x` in `sin(x)`, you place them inside the parentheses, and if it has no arguments, like `conjugate`, you simply provide the open and closing parentheses. `real`, `imag` and `conjugate` are attributes of the complex object, and `conjugate` is a *callable* attribute, known as a *method*.

OK, now you are an object oriented programmer. There are several key ideas in object oriented programming, and this is one of them: an object carries around with it data (simple attributes) and methods (callable attributes) that provide additional information about the object and perform services. It's one stop shopping – no need to go to external functions and libraries to deal with it – the object knows how to deal with itself.

## 2.3. Accessing the standard library

Arithmetic is fine, but before long you may find yourself tiring of it and wanting to compute logarithms and exponents, sines and cosines

```
>>> log(10)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'log' is not defined
```

These functions are not built into python, but don't despair, they are built into the python standard library. To access a function from the standard library, or an external library for that matter, you must import it.

```
>>> import math
>>> math.log(10)
2.3025850929940459
>>> math.sin(math.pi)
1.2246063538223773e-16
```

Note that the default `log` function is a base 2 logarithm (use `math.log10` for base 10 logs) and that floating point math is inherently imprecise, since analyticallysin$(\pi) = 0$.

It's kind of a pain to keep typing `math.log` and `math.sin` and `math.pi`, and python is accomodating. There are additional forms of `import` that will let you save more or less typing depending on your desires

```
# Appreviate the module name: m is an alias
>>> import math as m
>>> m.cos(2*m.pi)
1.0
```

```
# Import just the names you need
>>> from math import exp, log
>>> log(exp(1))
1.0
# Import everything - use with caution!
>>> from math import *
>>> sin(2*pi*10)
-2.4492127076447545e-15
```

To help you learn more about what you can find in the math library, python has nice introspection capabilities – introspection is a way of asking an object about itself. For example, to find out what is available in the math library, we can get a directory of everything available with the `dir` command[8]

```
>>> dir(math)
['__doc__', '__file__', '__name__', 'acos', 'asin', 'atan', 'atan2', 'ceil', 'cos', 'cosh', 'degrees', '
```

This gives us just a listing of the names that are in the math module – they are fairly self descriptive, but if you want more, you can call `help` on any of these functions for more information

```
>>> help(math.sin)
Help on built-in function sin:
sin(...)
sin(x)
Return the sine of x (measured in radians).
```

and for the whole math library

```
>>> help(math)
Help on module math:

NAME
    math

FILE
    /usr/local/lib/python2.3/lib-dynload/math.so

DESCRIPTION
    This module is always available.  It provides access to the
    mathematical functions defined by the C standard.

FUNCTIONS
    acos(...)
        acos(x)

        Return the arc cosine (measured in radians) of x.

    asin(...)
        asin(x)

        Return the arc sine (measured in radians) of x.
```

And much more which is snipped. Likewise, we can get information on the complex object in the same way

```
>>> x = complex(0,1)
>>> dir(x)
['__abs__', '__add__', '__class__', '__coerce__', '__delattr__', '__div__', '__divmod__', '__doc__', '__
```

Notice that called `dir` or `help` on the `math` *module*, the `math.sin` *function*, and the `complex` *number* `x`. That's because modules, functions and numbers are all *objects*, and we use the same object introspection and help capabilites on them. We can find out what type of object they are by calling `type` on them, which is another function in python's introspection arsenal

[8]In addition to the introdpection and help provided in the python interpreter, the official documentation of the python standard library is very good and up-to-date http://docs.python.org/lib/lib.html .

```
>>> type(math)
<type 'module'>
>>> type(math.sin)
<type 'builtin_function_or_method'>
>>> type(x)
<type 'complex'>
```

Now, you may be wondering: what were all those god-awful looking double underscore methods, like `__abs__` and `__mul__` in the `dir` listing of the complex object above? These are methods that define what it means to be a numeric type in python, and the complex object implements these methods so that complex numbers act like the way should, eg `__mul__` implements the rules of complex multiplication. The nice thing about this is that python specifies an application programming interface (API) that is the definition of what it means to be a number in python. And this means you can define your own numeric types, as long as you implement the required special double underscore methods for your custom type. double underscore methods are very important in python; although the typical newbie never sees them or thinks about them, they are there under the hood providing all the python magic, and more importantly, showing the way to let you make magic.

## 2.4. Strings

We've encountered a number of types of objects above: int, float, long, complex, method/function and module. We'll continue our tour with an introduction to strings, which are critical components of almost every program. You can create strings in a number of different ways, with single quotes, double quotes, or triple quotes – this diversity of methods makes it easy if you need to embed string characters in the string itself

```
# single, double and triple quoted strings
>>> s = 'Hi Mom!'
>>> s = "Hi Mom!"
>>> s = """Porky said, "That's all folks!" """
```

You can add strings together to concatenate them

```
# concatenating strings
>>> first = 'John'
>>> last = 'Hunter'
>>> first+last
'JohnHunter'
```

or call string methods to process them: upcase them or downcase them, or replace one character with another

```
# string methods
>>> last.lower()
'hunter'
>>> last.upper()
'HUNTER'
>>> last.replace('h', 'p')
'Hunter'
>>> last.replace('H', 'P')
'Punter'
```

Note that in all of these examples, the string `last` is unchanged. All of these methods operate on the string and return a new string, leaving the original unchanged. In fact, python strings cannot be changed by any python code at all: they are *immutable* (unchangeable). The concept of mutable and immutable objects in python is an important one, and it will come up again, because only immutable objects can be used as keys in python dictionaries and elements of python sets.

You can access individual characters, or slices of the string (substrings), using indexing. A string in sequence of characters, and strings implement the sequence protocol in python – we'll see more examples of python sequences later – and all sequences have the same syntax for accessing their elements. Python uses 0 based indexing which means the first element is at index 0; you can use negative indices to access the last elements in the sequence

```
# string indexing
>>> last = 'Hunter'
>>> last[0]
'H'
>>> last[1]
'u'
```

```
>>> last[-1]
'r'
```

To access substrings, or generically in terms of the sequence protocol, slices, you use a colon to indicate a range

```
# string slicing
>>> last[0:2]
'Hu'
>>> last[2:4]
'nt'
```

As this example shows, python uses "one-past-the-end" indexing when defining a range; eg, in the range `indmin:indmax`, the element of `imax` is not included. You can use negative indices when slicing too; eg, to get everything before the last character

```
>>> last[0:-1]
'Hunte'
```

You can also leave out either the min or max indicator; if they are left out, 0 is assumed to be the `indmin` and one past the end of the sequence is assumed to be `indmax`

```
>>> last[:3]
'Hun'
>>> last[3:]
'ter'
```

There is a third number that can be placed in a slice, a step, with syntax indmin:indmax:step; eg, a step of 2 will skip every second letter

```
>>> last[1:6:2]
'utr'
```

Although this may be more that you want to know about slicing strings, the time spent here is worthwhile. As mentioned above, all python sequences obey these rules. In addition to strings, lists and tuples, which are built-in python sequence data types and are discussed in the next section, the numeric arrays widely used in scientific computing also implement the sequence protocol, and thus have the same slicing rules.

EXERCISE 2.5. What would you expect last[:] to return?

One thing that comes up all the time is the need to create strings out of other strings and numbers, eg to create filenames from a combination of a base directory, some base filename, and some numbers. Scientists like to create lots of data files like and then write code to loop over these files and analyze them. We're going to show how to do that, starting with the newbie way and progressively building up to the way of python zen master. All of the methods below *work*, but the zen master way will more efficient, more scalable (eg to larger numbers of files) and cross-platform.[9] Here's the newbie way: we also introduce the for-loop here in the spirit of diving into python – note that python uses whitespace indentation to delimit the for-loop code block

```
# The newbie way
for i in (1,2,3,4):
    fname = 'data/myexp0' + str(i) + '.dat'
    print fname
```

Now as promised, this will print out the 4 file names above, but it has three flaws: it doesn't scale to 10 or more files, it is inefficient, and it is not cross platform. It doesn't scale because it hard-codes the '0' after `myexp`, it is inefficient because to add several strings requires the creation of temporary strings, and it is not cross-platform because it hard-codes the directory separator '/'.

```
# On the path to elightenment
for i in (1,2,3,4):
    fname = 'data/myexp%02d.dat'%i
    print fname
```

This example uses string interpolation, the funny % thing. If you are familiar with C programming, this will be no surprise to you (on linux/unix systems do `man sprintf` at the unix shell). The percent character is a string formatting character: `%02d` means to take an integer (the `d` part) and print it with two digits, padding zero on the left (the `%02` part). There is more to be said about string interpolation, but let's finish the job at hand. This example is better than the newbie way because is scales up to files numbered 0-99, and it is more efficient because it avoids the creation of temporary strings. For the platform independent part, we go to the

[9] "But it works" is a common defense of bad code; my rejoinder to this is "A computer scientist is someone who fixes things that aren't broken".

python standard library `os.path`, which provides a host of functions for platform-independent manipulations of filenames, extensions and paths. Here we use `os.path.join` to combine the directory with the filename in a platform independent way. On windows, it will use the windows path separator '\' and on unix it will use '/'.

```
# the zen master approach
import os
for i in (1,2,3,4):
    fname = os.path.join('data', 'myexp%02d.dat'%i)
    print fname
```

EXERCISE 2.6. Suppose you have data files named like

```
data/2005/exp0100.dat
data/2005/exp0101.dat
data/2005/exp0102.dat
...
data/2005/exp1000.dat
```

Write the python code that iterates over these files, constructing the filenames as strings in using `os.path.join` to construct the paths in a platform-independent way. *Hint*: read the help for `os.path.join`!

OK, I promised to torture you a bit more with string interpolation – don't worry, I remembered. The ability to properly format your data when printing it is crucial in scientific endeavors: how many signficant digits do you want, do you want to use integer, floating point representation or exponential notation? These three choices are provided with `%d`, `%f` and `%e`, with lots of variations on the theme to indicate precision and more

```
>>> 'warm for %d minutes at %1.1f C' % (30, 37.5)
'warm for 30 minutes at 37.5 C'
>>> 'The mass of the sun is %1.4e kg'% (1.98892*10**30)
'The mass of the sun is 1.9889e+30 kg'
```

There are two string methods, `split` and `join`, that arise frequenctly in Numeric processing, specifically in the context of processing data files that have comma, tab, or space separated numbers in them. `split` takes a single string, and splits it on the indicated character to a sequence of strings. This is useful to take a single line of space or comma separated values and split them into individual numbers

```
# s is a single string and we split it into a list of strings
# for further processing
>>> s = '1.0 2.0 3.0 4.0 5.0'
>>> s.split(' ')
['1.0', '2.0', '3.0', '4.0', '5.0']
```

The return value, with square brackets, indicates that python has returned a list of strings. These individual strings need further processing to convert them into actual floats, but that is the first step. The conversion to floats will be discussed in the next session, when we learn about list comprehensions. The converse method is join, which is often used to create string output to an ASCII file from a list of numbers. In this case you want to join a list of numbers into a single line for printing to a file. The example below will be clearer after the next section, in which lists are discussed

```
# vals is a list of floats and we convert it to a single
# space separated string
>>> vals = [1.0, 2.0, 3.0, 4.0, 5.0]
>>> ' '.join([str(val) for val in vals])
'1.0 2.0 3.0 4.0 5.0'
```

There are two new things in the example above. One, we called the join method directly on a string itself, and not on a variable name. Eg, in the previous examples, we always used the name of the object when accessing attributes, eg `x.real` or `s.upper()`. In this example, we call the `join` method on the string which is a single space. The second new feature is that we use a list comprehension `[str(val) for val in vals]` as the argument to `join`. `join` requires a sequence of strings, and the list comprehension converts a list of floats to a strings. This can be confusing at first, so don't dispair if it is. But it is worth bringing up early because list comprehensions are a very useful feature of python. To help elucidate, compare `vals`, which is a list of floats, with the conversion of `vals` to a list of strings using list comprehensions in the next line

```
# converting a list of floats to a list of strings
>>> vals
[1.0, 2.0, 3.0, 4.0, 5.0]
>>> [str(val) for val in vals]
```

```
['1.0', '2.0', '3.0', '4.0', '5.0']
```

### 2.7. The basic python data structures

Strings, covered in the last section, are sequences of characters. python has two additional built-in sequence types which can hold arbitrary elements: tuples and lists. tuples are created using parentheses, and lists are created using square brackets

```
# a tuple and a list of elements of the same type
# (homogeneous)
>>> t = (1,2,3,4)  # tuple
>>> l = [1,2,3,4]  # list
```

Both tuples and lists can also be used to hold elements of different types

```
# a tuple and list of int, string, float
>>> t = (1,'john', 3.0)
>>> l = [1,'john', 3.0]
```

Tuples and lists have the same indexing and slicing rules as each other, and as string discussed above, because both implement the python sequence protocol, with the only difference being that tuple slices return tuples (indicated by the parentheses below) and list slices return lists (indicated by the square brackets)

```
# indexing and slicing tuples and lists
>>> t[0]
1
>>> l[0]
1
>>> t[:-1]
(1, 'john')
>>> l[:-1]
[1, 'john']
```

So why the difference between tuples and lists? A number of explanations have been offered on the mailing lists, but the only one that makes a difference to me is that tuples are immutable, like strings, and hence can be used as keys to python dictionaries and included as elements of sets, and lists are mutable, and cannot. So a tuple, once created, can never be changed, but a list can. For example, if we try to reassign the first element of the tuple above, we get an error

```
>>> t[0] = 'why not?'
Traceback (most recent call last):
 File "<stdin>", line 1, in ?
TypeError: object doesn't support item assignment
```

But the same operation is perfectly accetable for lists

```
>>> l[0] = 'why not?'
>>> l
['why not?', 'john', 3.0]
```

lists also have a lot of methods, tuples have none, save the special double underscore methods that are required for python objects and sequences

```
# tuples contain only ■hidden■ double underscore methods
>>> dir(t)
['__add__', '__class__', '__contains__', '__delattr__', '__doc__', '__eq__', '__ge__', '__getattribute__
# but lists contain other methods, eg append, extend and
# reverse
>>> dir(l)
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__', '__delslice__', '__doc__', '__eq_
```

Many of these list methods change, or mutate, the list, eg append adds an element to the list: `extend` extends the list with a sequence of elements, `sort` sorts the list in place, `reverse` reverses it in place, `pop` takes an element off the list and returns it.

We've seen a couple of examples of creating a list above – let's look at some more using list methods

```
>>> x = []                    # create the empty list
>>> x.append(1)               # add the integer one to it
>>> x.extend(['hi', 'mom'])   # append two strings to it
>>> x
```

```
[1, 'hi', 'mom']
>>> x.reverse()              # reverse the list, in place
>>> x
['mom', 'hi', 1]
>>> len(x)
3
```

We mentioned list comprehensions in the last section when discussing string methods. List comprehensions are a way of creating a list using a for loop in a single line of python. Let's create a list of the perfect cubes from 1 to 10, first with a for loop and then with a list comprehension. The list comprehension code will not only be shorter and more elegant, it can be much faster (the dots are the indentation block indicator from the python shell and should not be typed)

```
# a list of perfect cubes using a for-loop
>>> cubes = []
>>> for i in range(1,10):
...     cubes.append(i**3)
...
>>> cubes
[1, 8, 27, 64, 125, 216, 343, 512, 729]
# functionally equivalent code using list comprehensions
>>> cubes = [i**3 for i in range(1,10)]
>>> cubes
[1, 8, 27, 64, 125, 216, 343, 512, 729]
```

The list comprehension code is faster because it all happens at the C level. In the simple for-loop version, the python expression which appends the cube of `i` has to be evaluated by the python interpreter for each element of the loop. In the list comprehension example, the single line is parsed once and executed at the C level. The difference in speed can be considerable, and the list comprehension example is shorter and more elegant to boot.

The remaining essential built-in data strucuture in python is the dictionary, which is an associative array that maps arbitrary immutable objects to arbitrary objects. int, long, float, string and tuple are all immutable and can be used as keys; to a dictionary list and dict are mutable and cannot. A dictionary takes one kind of object as the key, and this key points to another object which is the value. In a contrived but easy to comprehent examples, one might map names to ages

```
>>> ages = {}            # create an empty dict
>>> ages['john'] = 36
>>> ages['fernando'] = 33
>>> ages                 # view the whole dict
{'john': 36, 'fernando': 33}
>>> ages['john']
36
>>> ages['john'] = 37    # reassign john's age
>>> ages['john']
37
```

Dictionary lookup is very fast; Tim Peter's once joked that any python program which uses a dictionary is automatically 10 times faster than any C program, which is of course false, but makes two worthy points in jest: dictionary lookup is fast, and dictionaries can be used for important optimizations, eg, creating a cache of frequently used values. As a simple eaxample, suppose you needed to compute the product of two numbers between 1 and 100 in an inner loop – you could use a dictionary to cache the cube of all odd of numbers $< 100$; if you were inteterested in all numbers, you might simply use a list to store the cached cubes – I am cacheing only the odd numbers to show you how a dictionary can be used to represent a sparse data structure

```
>>> cubes = dict([ ( i, i**3 ) for i in range(1,100,2)])
>>> cubes[5]
125
```

The last example is syntactically a bit challenging, but bears careful study. We are initializing a dictionary with a list comprehension. The list comprehension is made up of length 2 tuples `( i, i**3 )`. When a dictionary is initialized with a sequence of length 2 tuples, it assumes the first element of the tuple `i` is the *key* and the second element i**3is the *value*. Thus we have a lookup table from odd integers to to cube. Creating dictionaries from list comprehensions as in this example is something that hard-core python programmers do almost every day, and you should too.

EXERCISE 2.8. Create a lookup table of the product of all pairs of numbers less than 100. The key will be a tuple of the two numbers `(i,j)` and the value will be the product. Hint: you can loop over multiple ranges in a list comprehension, eg `[ something for i in range(Ni) for j in range(Nj)]`

### 2.9. The Zen of Python

EXERCISE 2.10. `>>> import this`

### 2.11. Functions and classes

You can define functions just about anywhere in python code. The typical function definition takes zero or more arguments, zero or more keyword arguments, and is followed by a documentation string and the function definition, optionally returing a value. Here is a function to compute the hypoteneuse of a right triange

```
def hypot(base, height):
   'compute the hypoteneuse of a right triangle'
   import math
   return math.sqrt(base**2 + height**2)
```

As in the case of the for-loop, leading white space is significant and is used to delimt the start and end of the function. In the example below, x = 1 is not in the function, because it is not indented

```
def growone(l):
   'append 1 to a list l'
   l.append(1)
x = 1
```

Note that this function does not return anything, because the append method modifies the list that was passed in. You should be careful when designing functions that have side effects such as modifying the structures that are passed in; they should be named and documented in such a way that these side effects are clear.

Python is pretty flexible with functions: you can define functions within function definitions (just be mindful of your indentation), you can attach attributes to functions (like other objects), you can pass functions as arguments to other functions. A function keyword argument defines a default value for a function that can be overridden. Below is an example which provides a normalize keyword argument. The default argument is `normalize=None`; the value None is a standard python idiom which usually means either do the default thing or do nothing. If `normalize` is not `None`, we assume it is a function that can be called to normalize our data

```
def psd(x, normalize=None):
    'compute the power spectral density of x'
    if normalize is not None: x = normalize(x)
    # compute the power spectra of x and return it
```

This function could be called with or without a `normalize` keyword argument, since if the argument is not passed, the default of `None` is used and no normalization is done.

```
# no normalize argument; do the default thing
>>> psd(x)
# define a custom normalize function unitstd as pass it
# to psd
>>> def unitstd(x): return x/std(x)
>>> psd(x, normalize=unitstd)
```

In Section2.2 we noticed that complex objects have the real and imag data attributes, and the conjugate method. An object is an instance of a class that defines it, and in python you can easily define your own classes. In that section, we emphasized that one of the important features of a classes/objects is that they carry around their data and methods in a single bundle. Let's look at the mechnics of defining classes, and creating instances (a.k.a. objects) of these classes. Classes have a special double underscore method __init__ that is used as the function to initialize the class. For this example, we'll continue with the normalize theme above, but in this case the normalization requires some data parameters. This example arises when you want to normalize an image which may range over 0-255 (8 bit image) or from 0-65535 (16 bit image) to the 0-1 interval. For 16 bit images, you would normally divide everything by 65525, but you might want to configure this to a smaller number if your data doesn't use the whole intensity range to enhance contrast. For simplicitly, let's suppose our normalize class is only interested in the pixel maximum, and will divide all the data by that value.

```
from __future__ import division  # make sure we do float division
class Normalize:
    """
```

```
    A class to normalize data by dividing it by a maximum value
    """
    def __init__(self, maxval):
        'maxval will be mapped to 1'
        self.maxval = maxval
    def __call__(self, data):
        'do the normalization'
        # in real life you would also want to clip all values of
        # data>maxval so that the returned value will be in the unit
        # interval
        return data/self.maxval
```

The triple quoted string following the definition of class Normalize is the class documentation stringd, and it will bre shown to the user when they do `help(Normalize)`. A commonly used convention is to name classes with *UpperCase*, but this is not required. self is a special variable that a class can use to refer to its own data and methods, and must be the first argument to all the class methods. The `__init__` method stores the normalization value maxval as a class attribute in `self.maxval`, and this value can later be reused by other class methods (as it is in `__call__`) and it can be altered by the user of the class, as will illustrate below. The `__call__` method is another piece of python double underscore magic, it allows class instances to be used as *functions*, eg you can call them just like you can call any function. OK, now let's see how you could use this.

The first line use used to create an *instance* of the *class* `Normalize`, and the special method `__init__` is implicitly called. The second line implicitly calls the special `__call__`method

```
>>> norm = Normalize(65356) # good for 16 bit images
>>> norm(255)               # call this function
0.0039017075708427688
# We can reset the maxval attribute, and the call method
# is automagically updated
>>> norm.maxval = 255       # reset the maxval
>>> norm(255)               # and call it again
1.0
# We can pass the norm instance to the psd function we defined above, which
# is expecting a function
>>> pdf(X, normalize=norm)
```

EXERCISE 2.12. Pretend that `complex` were not built-in to the python core, and write your own complex class `MyComplex`. Provide `real` and `imag` attributes and the `conjugate` method. Define `__abs__`, `__mul__` and `__add__` to implement the absolute value of complex numbers, multiplication of complex numbers and addition of complex numbers. See the API definition of the python number protocol; although this is written for C programmers, it contains information about the required function call signatures for each of the double underscore methods that define the number protocol in python; where they use `o1` on that page, you would use `self` in python, and where they use `o2` you might use `other` in python.[10] To get you started, I'll show you what the `__add__` method should look like

```
# An example double underscore method required in your MyComplex
# implementation
def __add__(self, other):
    'add self to other and return a new MyComplex instance'
    r = self.real + other.real
    i = self.imag + other.imag
    return MyComplex(r,i)
# When you are finished, test your implementation with
>>> x = MyComplex(2,3)
>>> y = MyComplex(0,1)
>>> x.real
2.0
>>> y.imag
1.0
>>> x.conjugate()
(2-3j)
```

[10] http://www.python.org/doc/current/api/number.html

```
>>> x+y
(2+4j)
>>> x*y
(-3+2j)
>>> abs(x*y)
3.6055512754639891
```

### 2.13. Files and file like objects

Working with files is one of the most common and important things we do in scientific computing because that is usually where the data lives. In Section2.4, we went through the mechanics of automatically building file names like

```
data/myexp01.dat
data/myexp02.dat
data/myexp03.dat
data/myexp04.dat
```

but we didn't actually do anything with these files. Here we'll show how to read in the data and do something with it. Python makes working with files easy and dare I say fun. The test data set lives in `data/family.csv` and is a standard comma separated value file that contains information about my family: first name, last name, age, height in cm, weight in kg and birthdate. We'll open this file and parse it – note that python has a standard module for parsing CSV files that is much more sophisticated than what I am doing here. Nevertheless, it serves as an easy to understand example that is close enough to real life that it is worth doing. Here is what the data file looks like

```
First,Last,Age,Weight,Height,Birthday
John,Hunter,36,175,180,1968-03-05
Miriam,Sierig,33,135,177,1971-05-04
Rahel,Hunter,7,55,134,1998-02-25
Ava,Hunter,3,45,121,2001-04-26
Clara,Hunter,0,15,55,2004-10-02
```

Here is the code to parse that file

```
# open the file for reading
fh = file('../data/family.csv', 'r')
# slurp the header, splitting on the comma
headers = fh.readline().split(',')
# now loop over the remaining lines in the file and parse them
for line in fh:
    # remove any leading or trailing white space
    line = line.strip()
    # split the line on the comma into separate variables
    first, last, age, weight, height, dob = line.split(',')
    # convert some of these strings to floats
    age, weight, height = [float(val) for val in (age, weight, height)]
    print first, last, age, weight, height, dob
```

This example illustrates several interesting things. The syntax for opening a file is `file(filename, mode)` and the `mode` is a string like `'r'` or `'w'` that determines whether you are opening in read or write mode. You can also read and write binary files with `'rb'` and `'wb'`. There are more options and you should do `help(file)` to learn about them. We then use the file `readline` method to read in the first line of the file. This returns a string (the line of text) and we call the string method `split(',')` to split that string wherever it sees a comma, and this returns a list of strings which are the headers

```
>>> headers
['First', 'Last', 'Age', 'Weight', 'Height', 'Birthday\n']
```

The new line character `'\n'` at the end of `'Birthday\n'` indicates we forgot to strip the string of whitespace. To fix that, we should have done

```
>>> headers = fh.readline().strip().split(',')
>>> headers
['First', 'Last', 'Age', 'Weight', 'Height', 'Birthday']
```

Notice how this works like a pipeline: `fh.readline` returns a line of text as a string; we call the string method `strip` which returns a string with all white space (spaces, tabs, newlines) removed from the left and right; we then call the `split` method on this stripped string to split it into a list of strings.

Next we start to loop over the file – this is a nice feature of python file handles, you can iterate over them as a sequence. We've learned our lesson about trailing newlines, so we first strip the line with `line = line.strip()`. The rest is string processing, splitting the line on a comma as we did for the headers, and converting the strings to numbers where approriate by calling `float(val)` for each of `age`, `weight` and `height`. Notice how we use list comprehensions and tuple unpacking – the age, weight, `height = [float(val) for val in (age, weight, height)]` line, to convert several values at once.

Now that we have all this data, how mught we store it. We could store it in a `results` list

```
results = []
for line in fh:
    # process the line as above to get the variables
    results.append( (first, last, age, weight, height, dob) )
# and later when we want to analyze the data
for first, last, age, weight, height, dob in results:
    # do something with the data
```

EXERCISE 2.14. `zip` magic. Python has a nice funcion `zip` that lets you do very useful things with lists of tuples. `results` above is a list of tuples – each tuple is the `first`, `last`, `age`, `weight`, `height`, `dob` for a family member. What happens if you do

```
>>> first, last, age, weight, height, dob = zip(*results)
```

What is `age` now?

EXERCISE 2.15. Write a class `Person` and store the attributes `first`, `last`, `age`, `weight`, `height`, `dob` in that class. Add a class instance to the results list, eg

```
results.append(Person(first, last, age, weight, height, dob))
```

Python also has a special syntax for printing to an open writable file object

```
# open the file for writing
outfile = file('mydata.data', 'w')
for x,y,z in myresults:
    print >> outfile, '%1.3f %1.3f %1.3f'%(x,y,z)
```

Another really nice thing about file objects is that other classes can implement the file protcol and allow you to use them as if they were files. For example, the StringIO module in the standard library allows you to read and write to strings as if they were files. The urllib.urlopen function allows you to open a remove web page as a file object. Try this

```
# loop over the lines in google's html
from urllib import urlopen
for line in urlopen('http://www.google.com').readlines():
    print line,
```

CHAPTER 3

# A tour of IPython

One of Python's most useful features is its interactive interpreter. This system allows very fast testing of ideas without the overhead of creating test files as is typical in most programming languages. In scientific computing, one of the reasons behind the popularity of systems like Matlab ™, IDL ™or Mathematica ™, is precisely their interactive nature. Scientific computing is an inherently exploratory problem domain, where one is rarely faced with writing a program against a set of well-defined explicit constraints. Being able to load data, process it with different algorithms or test parameters, visualize it, save results, and do all of this in a fluid and efficient way, can make a big productivity difference in day to day scientific work. Even for the development of large codes, a good interactive interpreter can be a major asset, though this is a less commonly held view; later in this document we will discuss this aspect of the problem.

However, the interpreter supplied with the standard Python distribution is somewhat limited for extended interactive use. The IPython project [**30**] was born out of a desire to have a better Python interactive environment, which could combine the advantages of the Python language with some of the best ideas found in systems like IDL or Mathematica, along with many more enhancements. IPython is a free software project (released under the BSD license) which tries to:

(1) Provide an interactive shell superior to Python's default. IPython has many features for object introspection, system shell access, and its own special command system for adding functionality when working interactively. It tries to be a very efficient environment both for Python code development and for exploration of problems using Python objects (in situations like data analysis).
(2) Serve as an embeddable, ready to use interpreter for your own programs. IPython can be started with a single call from inside another program, providing access to the current namespace. This can be very useful both for debugging purposes and for situations where a blend of batch-processing and interactive exploration are needed.
(3) Offer a flexible framework which can be used as the base environment for other systems with Python as the underlying language. Specifically scientific environments like Mathematica, IDL and Matlab inspired its design, but similar ideas can be useful in many fields.

This document is not meant to replace the comprehensive IPython manual, which ships with the IPython distribution and is also available online at `http://ipython.scipy.org/doc/manual`. Instead, we will present here some relevant parts of it for everyday use, and refer readers to the full manual for in-depth details.

Additionally, this article by Jeremy Jones provides an introductory tutorial about IPython: `http://www.onlamp.com/pub/a/python/2005/01/27/ipython.html`.

## 3.1. Main IPython features

This section summarizes the most important user-visible features of IPython, which are not a part of the default Python shell or other interactive Python systems. While you can use IPython as a straight replacement for the normal Python shell, a quick read of these will allow you to take advantage of many enhancements which can be very useful in everyday work.

A bird's eye view of IPython's feature set:

- Dynamic object introspection. You can access docstrings, function definition prototypes, source code, source files and other details of any object accessible to the interpreter with a single keystroke ('`?`'). Adding a second `?` produces more details when possible.
- Completion in the local namespace, via the TAB key. This works for keywords, methods, variables and files in the current directory. TAB-completion, especially for attributes, is a convenient way to explore the structure of any object you're dealing with. Simply type object_name.<TAB> and a list of the object's attributes will be printed.
- Numbered input/output prompts with command history (persistent across sessions and tied to each profile), full searching in this history and caching of all input and output.

- User-extensible 'magic' commands. A set of commands prefixed with `%` is available for controlling IPython itself and provides directory control, namespace information and many aliases to common system shell commands.
- Alias facility for defining your own system aliases.
- Complete system shell access. Lines starting with ! are passed directly to the system shell, and using !! captures shell output into python variables for further use.
- The ability to expand python variables when calling the system shell. In a shell command, any python variable prefixed with `$` is expanded. A double `$$` allows passing a literal `$` to the shell (for access to shell and environment variables like `$PATH`).
- Filesystem navigation, via a magic `%cd` command, along with a persistent bookmark system (using `%bookmark`) for fast access to frequently visited directories.
- A macro system for quickly re-executing multiple lines of previous input with a single name, implemented via the `%macro` magic command.
- Session logging and restoring via the `%logstart`, `%logon/off` and `%logstate` magics. You can then later use these log files as code in your programs.
- Verbose and colored exception traceback printouts. Easier to parse visually, and in verbose mode they produce a lot of useful debugging information.
- Auto-parentheses: callable objects can be executed without parentheses: `'sin 3'` is automatically converted to `'sin(3)'`.
- Auto-quoting: using `','` as the first character forces auto-quoting of the rest of the line: `',my_function a b'` becomes automatically `'my_function("a","b")'`.
- Flexible configuration system. It uses a configuration file which allows permanent setting of all command-line options, module loading, code and file execution. The system allows recursive file inclusion, so you can have a base file with defaults and layers which load other customizations for particular projects.
- Embeddable. You can call IPython as a python shell inside your own python programs. This can be used both for debugging code or for providing interactive abilities to your programs with knowledge about the local namespaces (very useful in debugging and data analysis situations).
- Easy debugger access. You can set IPython to call up the Python debugger (pdb) every time there is an uncaught exception. This drops you inside the code which triggered the exception with all the data live and it is possible to navigate the stack to rapidly isolate the source of a bug. The `%run` magic command –with the `-d` option– can run any script under `pdb`'s control, automatically setting initial breakpoints for you.
- Profiler support. You can run single statements (similar to `profile.run()`) or complete programs under the profiler's control. While this is possible with the standard `profile` module, IPython wraps this functionality with magic commands (see `'%prun'` and `'%run -p'`) convenient for rapid interactive work.

## 3.2. Effective interactive work

IPython has been designed to try to make interactive work as fluid and efficient as possible. All of its features try to maximize the output-per-keystroke, so that as you work at an interactive console, minimal typing produces results. It makes extensive use of the readline library, has its own control system (magics), caches previous inputs and outputs, has a macro system, etc. Becoming familiar with these features, while not necessary for basic use, will make long-term use of the system much more pleasant and productive.

**3.2.1. Magic functions.** The default Python interactive shell only allows valid Python code to be typed at its input prompt. While this appears like a reasonable approach in principle, in practical use it turns out to be rather limiting. A good interactive environment should allow you to control the environment itself, in hopefully the most typing-efficient way.

Verbosity in code is a good thing, since code is a long-lived entity, and deciphering three-letter acronyms for variable names, 6 months after a program was written, is typically an exercise in frustration. However at an interactive prompt, where every keystroke counts and things are not meant to be permanent, compact and efficient control of your environment is an important feature. The default Python shell does not offer this, and the Python language's verbosity, which is an asset for the long-term readability of code, becomes a bit of a liability in this context.

For this reason, IPython offers a system of 'magic' commands, which serve to control IPython itself and perform a number of common tasks. Users of IDL will be familiar with the 'dot' commands, like `.stop`, which perform similar functions in that system. In IPython, the magic system covers much more functionality and

is fully user-extensible. This allows users to add all the control they may desire to their everyday working environment.

The magics system is patterned after the time-honored Unix shells, with whitespace separating arguments, no parentheses required, and dashes for specifying options to commands. Many builtin magics also are named like the Unix commands they mimic, so that an IPython environment can be used 'out of the box' by any Unix user with ease.

IPython will treat any line whose first character is a `%` as a special call to a magic function. For example: typing `'%cd mydir'` (without the quotes) changes you working directory to `'mydir'`, if it exists. For any magic function, typing its name followed by `?` will show you the magic's information and docstring, just like for other regular Python objects. Simply typing `magic` at the prompt will print an overview of the system, and a list of all the existing magics with their docstrings.

If you have 'automagic' enabled, you don't need to type in the `%` explicitly. Automagic is enabled by default, and you can configure this in your `ipythonrc` file, via the command line option `-automagic` or even toggle it at runtime with the `%automagic` function. IPython will scan its internal list of magic functions and call one if it exists. With automagic on you can then just type '`cd mydir`' to go to directory '`mydir`'. The automagic system has the lowest possible precedence in name searches, so defining an identifier with the same name as an existing magic function will shadow it for automagic use. You can still access the shadowed magic function by explicitly using the `%` character at the beginning of the line.

An example (with automagic on) should clarify all this:

```
In [1]: cd ipython # %cd is called by automagic
/home/fperez/ipython
In [2]: cd = 1 # now cd is just a variable
In [3]: cd .. # and doesn't work as a function anymore
------------------------------------------------------------
    File "<console>", line 1
      cd ..
         ^
SyntaxError: invalid syntax
In [4]: %cd .. # but %cd always works
/home/fperez
In [5]: del cd # if you remove the cd variable
In [6]: cd ipython # automagic can work again
/home/fperez/ipython
```

**3.2.2. Object exploration.** Python is a language with exceptional introspection capabilities. This means that, within the language itself, it is possible to extract a remarkable amount of information about all objects currently in memory. However the default Python shell exposes very little of this power in an easy to use manner; IPython provides a lot of functionality to remedy this.

The bulk of IPython's introspection system is accessible via only two keys: the question mark `?` and the `<TAB>` key. Under the hood, these two keys control a fairly complex set of libraries which ultimately rely on the `readline` and `inspect` modules from the Python standard library. But for regular use, you should never need to remember anything beyond these two. As an example, consider defining a variable named `mylist`, which starts as an empty list:

```
In [1]: mylist=[]
```

now you can find out some things about it by using the question mark:

```
In [2]: mylist?
Type:           list
Base Class:     <type 'list'>
String Form:    []
Namespace:      Interactive
Length:         0
Docstring:
    list() -> new list
    list(sequence) -> new list initialized from sequence's items
```

next, by adding a period (the standard Python attribute separator) and hitting `TAB`, IPython will show you all the attributes which this object has:

```
In [3]: mylist.<The TAB key was pressed here>
mylist.append   mylist.extend   mylist.insert   mylist.remove mylist.sort
```

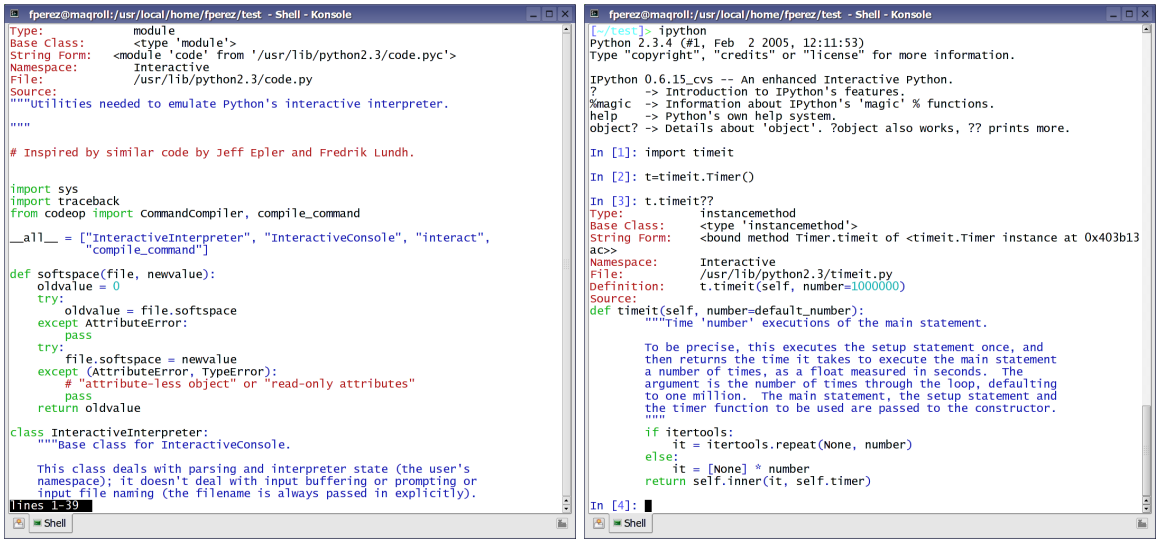



FIGURE 3.2.1. IPython can show syntax-highlighted source code for objects whose source is available.

```
mylist.count    mylist.index    mylist.pop      mylist.reverse
```

you can then request further details about any of them:

```
In [3]: mylist.append?
Type:           builtin_function_or_method
Base Class:     <type 'builtin_function_or_method'>
String Form:    <built-in method append of list object at 0x403b2b6c>
Namespace:      Interactive
Docstring:
    L.append(object) -- append object to end
```

The `?` system can be doubled. The first screenshot in Fig. 3.2.1 was generated by typing at the IPython prompt:

```
In [1]: import code
In [2]: code??
```

Using `??` shows the syntax-highlighted source for the `code` module from the Python standard library. This is an excellent way to explore modules or objects which you are not familiar with. As long as Python's `inspect` system is capable of finding the source code for an object, IPython will show it to you, with nice syntax highlights.

This can be done for entire modules, as in the prvious example, for individual functions, or even methods of object instances. The second screenshot in the same figure shows source for the `timeit` method of a `timeit.Timer` object.

The magic commands `%pdoc`, `%pdef`, `%psource` and `%pfile` will respectively print the docstring, function definition line, full source code and the complete file for any object (when they can be found).

**3.2.3. Input and Ouptut cached prompts.** In IPython, all output results are automatically stored in a global dictionary named `Out` and variables named `_1`, `_2`, etc. alias them. For example, the result of input line 4 is available either as `Out[4]` or as `_4`. Additionally, three variables named `_`, `__` and `___` are always kept updated with the for the last three results. This allows you to recall any previous result and further use it for new calculations. For example:

```
In [1]: 2+4
Out[1]: 6
In [2]: _+9
Out[2]: 15
In [3]: _+__
Out[3]: 21
In [4]: print _1
6
In [5]: print Out[1]
```

```
6
In [6]: _2**3
Out[6]: 3375
```

You can put a ';' at the end of a line to supress the printing of output. This is useful when doing calculations which generate long output you are not interested in seeing. The `_*` variables and the `Out[]` list do get updated with the contents of the output, even if it is not printed. You can thus still access the generated results this way for further processing.

A similar system exists for caching input. All input is stored in a global list called `In` , so you can re-execute lines 22 through 28 plus line 34 by typing `'exec In[22:29]+In[34]'` (using Python slicing notation).

At any time, your input history remains available. The `%hist` command can show you all previous input, without line numbers if desired (option `-n`) so you can directly copy and paste code either back in IPython or in a text editor. You can also save all your history by turning on logging via `%logstart`; these logs can later be either reloaded as IPython sessions or used as code for your programs.

If you need to execute the same set of lines often, you can assign them to a macro with the `%macro` magic function. Macros are simply short names for groups of input lines, which can be re-executed by only typing that name. Typing `macro?` at the prompt will show you the function's full documentation. For example, if your history contains:

```
44: x=1
45: y=3
46: z=x+y
47: print x
48: a=5
49: print 'x',x,'y',y
```

You can create a macro with lines 44 through 47 (included) and line 49 called `my_macro` with:

```
In [51]: %macro my_macro 44:48 49
```

Now, simply typing `my_macro` will re-execute all this code in one pass. The number range follows standard Python list slicing notation, where `n:m` means the numbers $(n, n+1, \ldots, m-1)$.

You should note that macros execute in the current context, so if any variable changes, the macro will pick up the new value every time it is executed:

```
In [1]: x=1
In [2]: y=x*5
In [3]: z=x+3
In [4]: print 'y is:',y,'and z is:',z
y is: 5 and z is: 4
# make a macro with lines 2,3,4 (note Python list slice syntax):
In [5]: macro yz 2:5
Macro `yz` created. To execute, type its name (without quotes).
Macro contents:
y=x*5
z=x+3
print 'y is:',y,'and z is:',z
# now, run the macro directly:
In [6]: yz
Out[6]: Executing Macro...
y is: 5 and z is: 4
# we change the value of x
In [7]: x=9
# and now if we rerun the macro, we get the new values:
In [8]: yz
Out[8]: Executing Macro...
y is: 45 and z is: 12
```

**3.2.4. Running code.** The `%run` magic command allows you to run any python script and load all of its data directly into the interactive namespace. `%run` is a sophisticated wrapper around the Python `execfile()` builtin function; since the file is re-read from disk each time, changes you make to it are reflected immediately (in contrast to the behavior of `import`). I rarely use `import` for code I am testing, relying on `%run` instead.

By default,

```
%run myfile arg1 arg2 ...
```

executes `myfile` in a namespace initially consisting only of `__name__=='__main__'` and `sys.argv` being filled with arg1, arg2, etc. This means that using `%run` is functionally very simlar to executing a script at the system command line, but you get all the functionality of IPython (better tracebacks, debugger and profiler access, etc.). The `-n` option prevents `__name__` from being set equal to `'__main__'`, in case you want to test the part of a script which only runs when `import`ed.

Additionally, the fact that IPython then updates your interactive namespace with the variables defined in the script is very useful, because you can run your code to do a lot of processing, and then continue using and exploring interactively the objects created by the program.

For example, if the file `ip_simple.py` contains:

```
import sys
print 'sys.argv is:',sys.argv
print '__name__ is:',__name__
x = 1
```

you can run it in IPython as follows:

```
# First, let's check that x is undefined
In [1]: x
---------------------------------------------------------------------------
exceptions.NameError                                 Traceback (most recent call last)
/usr/local/home/fperez/teach/course/examples/<console>
NameError: name 'x' is not defined
# Now we run the script (the .py extension is optional):
In [2]: run ip_simple
sys.argv is: ['ip_simple.py']
__name__ is: __main__
# If we print x, now it has the value from the script
In [3]: x
Out[3]: 1
# Again, but now running with some arguments:
In [4]: run ip_simple -x arg1 "hello world"
sys.argv is: ['ip_simple.py', '-x', 'arg1', 'hello world']
__name__ is: __main__
```

With the `-i` option, the namespace where your script runs is actually your interactive one. This can be used for two sligthly different purposes. The simpler case, is just to quickly type up a set of commands in an editor which you want to execute on your current environment (although the `%edit` command can also be used for this). Consider running the file `ip_simple2.py`:

```
"""This simple file prints a variable which is NOT defined here.
It should be run via IPython's %run with the -i option."""
print 'x is:',x
```

in IPython:

```
# A regular %run will produce an error:
In [1]: run ip_simple2
---------------------------------------------------------
exceptions.NameError              Traceback (most recent call last)
/usr/local/home/fperez/teach/course/examples/ip_simple2.py
      2
      3 It should be run via IPython's %run with the -i option."""
      4
----> 5 print 'x is:',x
      6
NameError: name 'x' is not defined
WARNING: Failure executing file: <ip_simple2.py>
x is:
# However, if you do have a variable x defined:
In [2]: x='hello'
# you can use the -i option and the code will see x:
In [3]: run -i ip_simple2
```

```
x is: hello
```

A different use of `%run -i`, is to repeatedly run scripts which may have a potentially expensive initialization phase. If this initialization does not need to be repeated on each run (for example, you are debugging some other submodule and can reuse the same expensive object several times), you can avoid it by protecting the expensive object with a `try/except` block. This simple script illustrates the technique:

```
"""Example script with an expensive initialization.

Meant to be used via ipython's %run -i, though it can run standalone."""

# Imagine that bigobject is actually something whose creation is an expensive
# process, though here we are just going to make it a list of numbers for
# demonstration's sake.  The trick is to trap a test for the existence of this
# name in a try/except block.  If the object exists, we don't recreate it, if
# it doesn't exist yet (such as the first time the code is run in any given
# session), we make it.

try:
    bigobject
    print "We found bigobject! No need to initialize it."
except NameError:
    print "bigobject not found, performing expensive initialization..."
    bigobject = range(1000)

# And now you can move on with working on bigobject:
total = sum(bigobject)
print 'total is:',total
```

In IPython, here is how you can use it:

```
# The first time it runs, it will have to initialize
In [1]: run -i ip_expensive_init.py
bigobject not found, performing expensive initialization...
total is: 499500
# but successive runs don't require initialization
In [2]: run -i ip_expensive_init.py
We found bigobject! No need to initialize it.
total is: 499500
# you can still run without -i, to achieve a full reload
# if you need it for any reason
In [3]: run ip_expensive_init.py
bigobject not found, performing expensive initialization...
total is: 499500
```

In the third run, by not using `-i`, your script runs in an empty namespace and this forces a full initialization (the `NameError` exception is triggered).

`%run` also has special flags for timing the execution of your scripts (`-t`) and for executing them under the control of either Python's `pdb` debugger (`-d`) or profiler (`-p`). You can get all of its docstring with the usual `run?` mechanism.

Thanks to all of its various control options, `%run` can be used as the main tool for efficient interactive development of code which you write in your editor of choice. My personal operation mode, which has served me well for several years of scientific work in Python, is to have a good editor (XEmacs in my case) open with all my Python code, and IPython open in a terminal where I run, debug, explore, plot, etc.

## 3.3. Access to the underlying Operating System

**3.3.1. Basic usage.** IPython allows you to always access the underlying OS very easily. Any lines starting with `!` are passed directly to the system shell:

```
In [6]: !ls ip*.py
ip_expensive_init.py  ip_simple2.py  ip_simple.py
```

and using `!!` captures shell output into python variables for further use:

```
In [7]: !!ls ip*.py
Out[7]: ['ip_expensive_init.py', 'ip_simple2.py', 'ip_simple.py']
```

There is a difference between the two cases: in the first, the `ls` command simply prints its results to the terminal as text, but no value is returned. In the second, IPython actually captures the output of the command, splits it as a list (one line per entry), and returns its value. This allows you to then operate on the results with Python routines.

Additionally, IPython plays a few interesting syntactic tricks for your convenience. Whenever you make a system call, IPython will expand any call of the type `$var` into the actual value of the python variable `var`, so that you can call shell commands on Python values. Continuing the session above, and remembering that `_` holds the previously returned value, we can call the '`wc -l`' Unix command (which does a line count on a file) on the files we just obtained:

```
In [8]: for f in _:
   ...:      if 'simple' in f:
   ...:          !wc -l $f
   ...:
3 ip_simple2.py
4 ip_simple.py
```

While this is completely unorthodox (actually, invalid) Python, it is the kind of functionality which can make for extremely efficient uses when working at an interactive command line. Obviously all of this can be done (and it *is* done that way by IPython internally) with regular Python code, but that approach requires a fair amount more typing, the use of `%`-based string interpolation, and making system calls via the `os.system()` function.

If you actually need to pass a `$` character to a shell command, you simply use `$$` in the IPython command line:

```
In [11]: !echo $$SHELL
/bin/tcsh
```

If you want to capture the output of a system command directly to a named Python variable, you can use the `%sc` magic function:

```
# by default, %sc captures to a plain string:
In [16]: %sc astr=ls ip*.py
In [17]: astr
Out[17]: 'ip_expensive_init.py\nip_simple2.py\nip_simple.py'
# but with the -l option, it splits to a list (like !! does)
In [18]: %sc -l alist=ls ip*.py
In [19]: alist
Out[19]: ['ip_expensive_init.py', 'ip_simple2.py', 'ip_simple.py']
```

**3.3.2. System aliases.** In IPython, you can also define your own system aliases. Even though IPython gives you access to your system shell via the `!` prefix, it is convenient to have aliases to the system commands you use most often. This allows you to work seamlessly from inside IPython with the same commands you are used to in your system shell:

'`%alias alias_name cmd`' defines '`alias_name`' as an alias for '`cmd`'

Then, typing '`alias_name params`' will execute the system command '`cmd params`' (from your underlying operating system). Aliases have lower precedence than magic functions and Python normal variables, so if '`foo`' is both a Python variable and an alias, the alias can not be executed until '`del foo`' removes the Python variable. If you need to access an alias directly, you can use the builtin function `ipalias` as `ipalias('foo')`.

You can use the `%l` specifier in an alias definition to represent the whole line when the alias is called. For example:

```
In [2]: alias all echo "Input in brackets: <%l>"
In [3]: all hello world
Input in brackets: <hello world>
```

You can also define aliases with positional parameters using `%s` specifiers (one per parameter):

```
In [1]: alias parts echo first %s second %s
In [2]: %parts A B
first A second B
In [3]: %parts A
Incorrect number of arguments: 2 expected.
```

```
parts is an alias to: 'echo first %s second %s'
```

Aliases expand Python variables just like system calls using `!` or `!!` do: all expressions prefixed with '`$`' get expanded. For details of the semantic rules, see PEP-215: `http://www.python.org/peps/pep-0215.html`. This is the library used by IPython for variable expansion.

Simply typing `alias` will print a list of the current aliases, and `unalias` can be used to remove an alias. For further details, use `alias?`.

**3.3.3. Directory management.** IPython comes with some pre-defined aliases and a complete system for changing directories, both via a stack (see `%pushd`, `%popd` and `%ds`) and via direct `%cd`. The latter keeps a history of visited directories and allows you to go to any previously visited one. You can see this history with the `%dhist` magic:

```
In [1]: cd ~/code/python
/home/fperez/code/python
In [2]: cd ~/teach/
/home/fperez/teach
In [3]: cd ~/research
/home/fperez/research
In [4]: dhist
Directory history (kept in _dh)
0: /home/fperez/teach/course/examples
1: /home/fperez/code/python
2: /home/fperez/teach
3: /home/fperez/research
In [5]: cd -1
/home/fperez/code/python
```

The `%bookmark` magic allows you to create named bookmarks in your filesystem, which `cd` can be directed to go to (with the `-b` flag), and to which it will try to default automatically if no such named directory exists. The system is very easy to use and quite natural in practice:

```
In [8]: bookmark course
In [9]: cd
/home/fperez
In [10]: ls course
ls: course: No such file or directory
In [11]: cd course
(bookmark:course) -> /home/fperez/teach/course
/home/fperez/teach/course
```

**3.3.4. IPython as a system shell.** While IPython is *not* a system shell, it ships with a special profile called `pysh`, which you can activate at the command line as '`ipython -p pysh`'. This modifies IPython's behavior and adds some additional facilities and a prompt customized for filesystem navigation.

Note that this does *not* make IPython a full-fledged system shell. In particular, it has no job control, so if you type Ctrl-Z (under Unix), you'll suspend pysh itself, not the process you just started.

What the shell profile allows you to do is to use the convenient and powerful syntax of Python to do quick scripting at the command line. Below we describe some of its features.

3.3.4.1. *Aliases.* All of your `$PATH` has been loaded as IPython aliases, so you should be able to type any normal system command and have it executed. See `%alias?` and `%unalias?` for details on the alias facilities. See also `%rehash?` and `%rehashx?` for details on the mechanism used to load `$PATH`.

3.3.4.2. *Special syntax.* Any lines which begin with '`~`', '`/`' and '`.`' will be executed as shell commands instead of as Python code. The special escapes below are also recognized. `!cmd` is valid in single or multi-line input, all others are only valid in single-line input:

`!cmd:` pass 'cmd' directly to the shell
`!!cmd:` execute 'cmd' and return output as a list (split on '\n')
`$var=cmd:` capture output of cmd into var, as a string (shorthand for `%sc var=cmd`)
`$$var=cmd:` capture output of cmd into var, as a list (split on '\n', shorthand for `%sc -l var=cmd`)

3.3.4.3. *Useful functions and modules.* The os, sys and shutil modules from the Python standard library are automatically loaded. Some additional functions, useful for shell usage, are listed below. You can request more help about them with '`?`'.

`shell:` - execute a command in the underlying system shell

`system:` - like `shell()`, but return the exit status of the command
`sout:` - capture the output of a command as a string
`lout:` - capture the output of a command as a list (split on '\n')
`getoutputerror:` - capture (output,error) of a shell commandss

`sout/lout` are the functional equivalents of `$/$$`. They are provided to allow you to capture system output in the middle of true python code, function definitions, etc (where `$` and `$$` are invalid)

### 3.4. Access to an editor

You can use `%edit` to have almost multiline editing. While IPython doesn't support true multiline editing, this command allows you to call an editor on the spot, and IPython will execute the code you type in there as if it were typed interactively.

`%edit` runs your IPython configured editor. By default this is read from your environment variable `$EDITOR`. If this isn't found, it will default to `vi` under Linux/Unix and to `notepad` under Windows.

You can also set the value of this editor via the command-line option `'-editor'` or in your `ipythonrc` file. This is useful if you wish to use specifically for IPython an editor different from your typical default (and for Windows users who typically don't set environment variables).

This command allows you to conveniently edit multi-line code right in your IPython session.

If called without arguments, `%edit` opens up an empty editor with a temporary file and will execute the contents of this file when you close it (don't forget to save it!).

### 3.5. Customizing IPython

**3.5.1. Basics.** IPython has a very flexible configuration system. It uses a configuration file which allows permanent setting of all command-line options, module loading, code and file execution. The system allows recursive file inclusion, so you can have a base file with defaults and layers which load other customizations for particular projects.

IPython reads a configuration file which can be specified at the command line (`-rcfile`) or which by default is assumed to be called `ipythonrc`. Such a file is looked for in the current directory where IPython is started and then in your `IPYTHONDIR`, which allows you to have local configuration files for specific projects. The default value for this directory is `$HOME/.ipython` (`_ipython` under Windows). Under Unix operating systems `$HOME` always exists; for Windows, IPython will try to find such an environment variable; if it doesn't exist, it uses `HOMEDRIVE\HOMEPATH` (these are always defined by Windows). This typically gives something like `C:\Documents and Settings\YourUserName`, but your local details may vary. Finally, you can make this directory live anywhere you want by creating an environment variable called `$IPYTHONDIR`.

In this directory you will find all the files that configure IPython's defaults, and you can put there your profiles and extensions. This directory is automatically added by IPython to `sys.path`, so anything you place there can be found by `import` statements.

The syntax of an rcfile is one of key-value pairs separated by whitespace, one per line. Lines beginning with a `#` are ignored as comments, but comments can **not** be put on lines with data (the parser is fairly primitive). You can study the default rcfile created by IPython at startup for customization details, it is extremely commented.

**3.5.2. Profiles.** IPython can load any configuration file you want if you give its name at startup with the `-rcfile` flag. However, for convenience it provides a shorthand based on a naming convention for loading such profiles. This system allows you to easily maintain customized versions of IPython for specific purposes.

With the `-profile <name>` flag (you can abbreviate it to `-p`), IPython will assume that your config file is called `ipythonrc-<name>` (it looks in current dir first, then in `IPYTHONDIR`). This is a quick way to keep and load multiple config files for different tasks, especially if you use the include option of config files. You can keep a basic `IPYTHONDIR/ipythonrc` file and then have other profiles which include this one and load extra things for particular tasks. For example:

(1) `$HOME/.ipython/ipythonrc`: load basic things you always want.
(2) `$HOME/.ipython/ipythonrc-math`: load (1) and basic math-related modules.
(3) `$HOME/.ipython/ipythonrc-numeric`: load (1) and Numeric and plotting modules.

Since it is possible to create an endless loop by having circular file inclusions, IPython will stop if it reaches 15 recursive inclusions.

### 3.6. Debugging and profiling with IPython

The Python standard library includes powerful facilities for debugging and profiling code, but it is common to find even experienced Python programmers who still do not take advantage of them. In part, this is due to the fact that loading and configuring them requires reading an extra documentation section, and keeping a bit

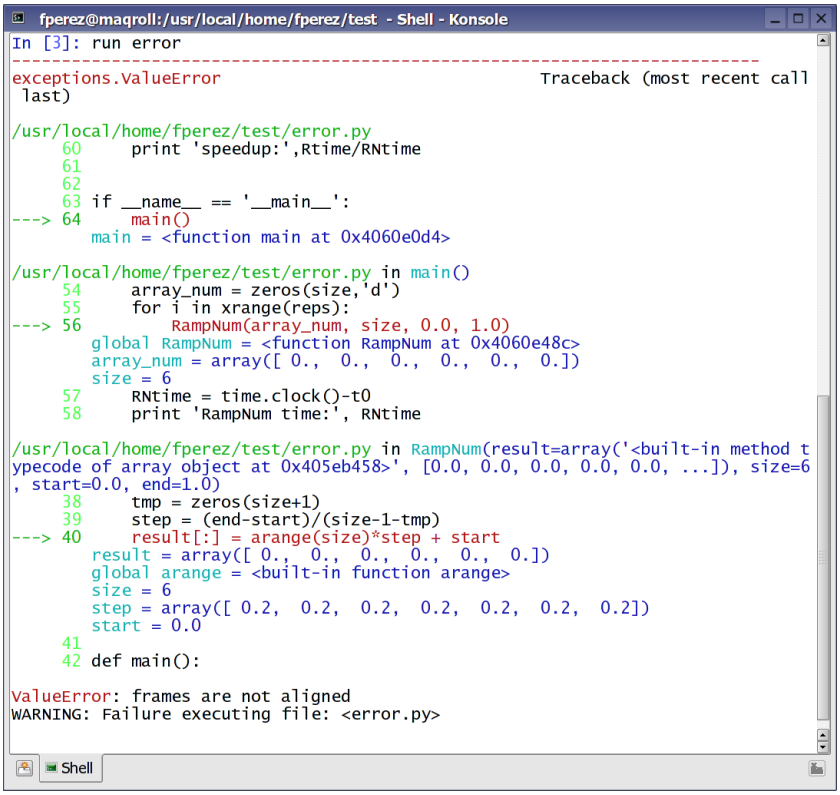



FIGURE 3.6.1. IPython can provide extremely detailed tracebacks.

of additional information about their use in your head. IPython tries to automate their use to the point where, with a single command, you can use either of these subsystems in a transparent manner. Hopefully they will become part of your daily workflow.

At its most basic, for debugging your programs, you can rely on using `%run` to execute them, see the results, play with all variables loaded into the interactive namespace, etc. A typical working session involves keeping your favorite editor open with the file you are working on, and repeatedly calling `%run` on it as you make changes and save them.

If your program raises an exception, IPython will provide you with a more detailed traceback than the default Python ones. You can even increase the level of detail further by using `%xmode Verbose`, which forces the printing of variable values at all stack frames. This option should be used with care though (and that's why it is not the default), as printing a ten-million-entry array can lock up your computer for a very long time. An example of this kind of very informative traceback is shown in Fig. 3.6.1.

**3.6.1. Automatic invocation of `pdb` on exceptions.** IPython, if started with the `-pdb` option (or if the option is set in your rc file) can call the Python `pdb` debugger every time your code triggers an uncaught exception. This feature can also be toggled at any time with the `%pdb` magic command. This can be extremely useful in order to find the origin of subtle bugs, because `pdb` opens up at the point in your code which triggered the exception, and while your program is at this point 'dead', all the data is still available and you can walk up and down the stack frame and understand the origin of the problem.

Furthermore, you can use these debugging facilities both with the embedded IPython mode and without IPython at all. For an embedded shell (see sec. 3.7), simply call the constructor with `'-pdb'` in the argument string and automatically `pdb` will be called if an uncaught exception is triggered by your code.

For stand-alone use of the feature in your programs which do not use IPython at all, put the following lines toward the top of your 'main' routine:

```
import sys,IPython.ultraTB
sys.excepthook = IPython.ultraTB.FormattedTB(mode='Verbose',
color_scheme='Linux', call_pdb=1)
```

The `mode` keyword can be either `'Verbose'` or `'Plain'`, giving either very detailed or normal tracebacks respectively. The `color_scheme` keyword can be one of `'NoColor'`, `'Linux'` (default) or `'LightBG'`. These are the same options which can be set in IPython with `-colors` and `-xmode`.

This will give any of your programs detailed, colored tracebacks with automatic invocation of `pdb`.

**3.6.2. Running entire programs via `pdb`.** `pdb`, the Python debugger, is a powerful interactive debugger which allows you to step through code, set breakpoints, watch variables, etc. IPython makes it very easy to start any script under the control of `pdb`, regardless of whether you have wrapped it into a `'main()'` function or not. For this, simply type `'%run -d myscript'` at an IPython prompt. See the `%run` command's documentation (`run?`) for more details, including how to control where `pdb` will stop execution first.

For more information on the use of the `pdb` debugger, read the included `pdb.doc` file (part of the standard Python distribution). On a stock Linux system it is located at `/usr/lib/python2.3/pdb.doc`, but the easiest way to read it is by using the `help()` function of the `pdb` module as follows (in an IPython prompt):

```
In [1]: import pdb
In [2]: pdb.help()
```

This will load the `pdb.doc` document in a file viewer for you automatically.

**3.6.3. Profiling.** When dealing with performance issues, the `%run` command with a `-p` option allows you to run complete programs under the control of the Python profiler. The `%prun` command does a similar job for single Python expressions (like function calls, similar to `profile.run()`). While this is possible with the standard `profile` module, IPython wraps this functionality with magic commands convenient for rapid interactive work.

## 3.7. Embedding IPython into your programs

A few lines of code are enough to load a complete IPython inside your own programs, giving you the ability to work with your data interactively after automatic processing has been completed.

You can call IPython as a python shell inside your own python programs. This can be used both for debugging code or for providing interactive abilities to your programs with knowledge about the local namespaces (very useful in debugging and data analysis situations).

It is possible to start an IPython instance *inside* your own Python programs. This allows you to evaluate dynamically the state of your code, operate with your variables, analyze them, etc. Note however that any changes you make to values while in the shell do *not* propagate back to the running code, so it is safe to modify your values because you won't break your code in bizarre ways by doing so.

This feature allows you to easily have a fully functional python environment for doing object introspection anywhere in your code with a simple function call. In some cases a simple print statement is enough, but if you need to do more detailed analysis of a code fragment this feature can be very valuable.

It can also be useful in scientific computing situations where it is common to need to do some automatic, computationally intensive part and then stop to look at data, plots, etc[1]. Opening an IPython instance will give you full access to your data and functions, and you can resume program execution once you are done with the interactive part (perhaps to stop again later, as many times as needed).

The following code snippet is the bare minimum you need to include in your Python programs for this to work (detailed examples follow later):

```
from IPython.Shell import IPShellEmbed
ipshell = IPShellEmbed()
ipshell() # this call anywhere in your program will start IPython
```

You can run embedded instances even in code which is itself being run at the IPython interactive prompt with '`%run <filename>`'. Since it's easy to get lost as to where you are (in your top-level IPython or in your embedded one), it's a good idea in such cases to set the in/out prompts to something different for the embedded instances. The code examples below illustrate this.

You can also have multiple IPython instances in your program and open them separately, for example with different options for data presentation. If you close and open the same instance multiple times, its prompt counters simply continue from each execution to the next.

Please look at the docstrings in the `Shell.py` module for more details on the use of this system.

The following sample file illustrating how to use the embedding functionality is provided in the examples directory as `example-embed.py`. It should be fairly self-explanatory:

[1]This functionality was inspired by IDL's combination of the `stop` keyword and the `.continue` executive command, which I have found very useful in the past, and by a posting on comp.lang.python by cmkl <cmkleffner@gmx.de> on Dec. 06/01 concerning similar uses of pyrepl.

```
#!/usr/bin/env python

"""An example of how to embed an IPython shell into a running program.

Please see the documentation in the IPython.Shell module for more details.

The accompanying file example-embed-short.py has quick code fragments for
embedding which you can cut and paste in your code once you understand how
things work.

The code in this file is deliberately extra-verbose, meant for learning."""

# The basics to get you going:

# IPython sets the __IPYTHON__ variable so you can know if you have nested
# copies running.

# Try running this code both at the command line and from inside IPython (with
# %run example-embed.py)
try:
    __IPYTHON__
except NameError:
    nested = 0
    args = ['']
else:
    print "Running nested copies of IPython."
    print "The prompts for the nested copy have been modified"
    nested = 1
    # what the embedded instance will see as sys.argv:
    args = ['-pi1','In <\\#>:','-pi2','   .\\D.:','-po','Out<\\#>:','-nosep']

# First import the embeddable shell class
from IPython.Shell import IPShellEmbed

# Now create an instance of the embeddable shell. The first argument is a
# string with options exactly as you would type them if you were starting
# IPython at the system command line. Any parameters you want to define for
# configuration can thus be specified here.
ipshell = IPShellEmbed(args,
                       banner = 'Dropping into IPython',
                       exit_msg = 'Leaving Interpreter, back to program.')

# Make a second instance, you can have as many as you want.
if nested:
    args[1] = 'In2<\\#>'
else:
    args = ['-pi1','In2<\\#>:','-pi2','   .\\D.:','-po','Out<\\#>:','-nosep']
ipshell2 = IPShellEmbed(args,banner = 'Second IPython instance.')

print '\nHello. This is printed from the main controller program.\n'

# You can then call ipshell() anywhere you need it (with an optional
# message):
ipshell('***Called from top level. '
```

```
        'Hit Ctrl-D to exit interpreter and continue program.')

print '\nBack in caller program, moving along...\n'

#---------------------------------------------------------------------------
# More details:

# IPShellEmbed instances don't print the standard system banner and
# messages. The IPython banner (which actually may contain initialization
# messages) is available as <instance>.IP.BANNER in case you want it.

# IPShellEmbed instances print the following information everytime they
# start:

# - A global startup banner.

# - A call-specific header string, which you can use to indicate where in the
# execution flow the shell is starting.

# They also print an exit message every time they exit.

# Both the startup banner and the exit message default to None, and can be set
# either at the instance constructor or at any other time with the
# set_banner() and set_exit_msg() methods.

# The shell instance can be also put in 'dummy' mode globally or on a per-call
# basis. This gives you fine control for debugging without having to change
# code all over the place.

# The code below illustrates all this.


# This is how the global banner and exit_msg can be reset at any point
ipshell.set_banner('Entering interpreter - New Banner')
ipshell.set_exit_msg('Leaving interpreter - New exit_msg')

def foo(m):
    s = 'spam'
    ipshell('***In foo(). Try @whos, or print s or m:')
    print 'foo says m = ',m

def bar(n):
    s = 'eggs'
    ipshell('***In bar(). Try @whos, or print s or n:')
    print 'bar says n = ',n

# Some calls to the above functions which will trigger IPython:
print 'Main program calling foo("eggs")\n'
foo('eggs')

# The shell can be put in 'dummy' mode where calls to it silently return. This
# allows you, for example, to globally turn off debugging for a program with a
# single call.
ipshell.set_dummy_mode(1)
```

```
print '\nTrying to call IPython which is now "dummy":'
ipshell()
print 'Nothing happened...'
# The global 'dummy' mode can still be overridden for a single call
print '\nOverriding dummy mode manually:'
ipshell(dummy=0)

# Reactivate the IPython shell
ipshell.set_dummy_mode(0)

print 'You can even have multiple embedded instances:'
ipshell2()

print '\nMain program calling bar("spam")\n'
bar('spam')

print 'Main program finished. Bye!'

#********************** End of file <example-embed.py> **********************
```

Once you understand how the system functions, you can use the following code fragments in your programs which are ready for cut and paste:

```
"""Quick code snippets for embedding IPython into other programs.

See example-embed.py for full details, this file has the bare minimum code for
cut and paste use once you understand how to use the system."""

#-----------------------------------------------------------------------------
# This code loads IPython but modifies a few things if it detects it's running
# embedded in another IPython session (helps avoid confusion)

try:
    __IPYTHON__
except NameError:
    argv = ['']
    banner = exit_msg = ''
else:
    # Command-line options for IPython (a list like sys.argv)
    argv = ['-pi1','In <\\#>:','-pi2','   .\\D.:','-po','Out<\\#>:']
    banner = '*** Nested interpreter ***'
    exit_msg = '*** Back in main IPython ***'

# First import the embeddable shell class
from IPython.Shell import IPShellEmbed
# Now create the IPython shell instance. Put ipshell() anywhere in your code
# where you want it to open.
ipshell = IPShellEmbed(argv,banner=banner,exit_msg=exit_msg)

#-----------------------------------------------------------------------------
# This code will load an embeddable IPython shell always with no changes for
# nested embededings.

from IPython.Shell import IPShellEmbed
ipshell = IPShellEmbed()
```

```
# Now ipshell() will open IPython anywhere in the code.

#-----------------------------------------------------------------------
# This code loads an embeddable shell only if NOT running inside
# IPython. Inside IPython, the embeddable shell variable ipshell is just a
# dummy function.

try:
    __IPYTHON__
except NameError:
    from IPython.Shell import IPShellEmbed
    ipshell = IPShellEmbed()
    # Now ipshell() will open IPython anywhere in the code
else:
    # Define a dummy ipshell() so the same code doesn't crash inside an
    # interactive IPython
    def ipshell(): pass

#****************** End of file <example-embed-short.py> ******************
```

### 3.8. Integration with Matplotlib

The matplotlib library (`http://matplotlib.sourceforge.net`) provides high quality 2D plotting for Python. Matplotlib can produce plots on screen using a variety of GUI toolkits, including Tk, GTK and WXPython. It also provides a number of commands useful for scientific computing, all with a syntax compatible with that of the popular Matlab program.

IPython accepts the special option `-pylab`. This configures it to support matplotlib, honoring the settings in the `.matplotlibrc` file. IPython will detect the user's choice of matplotlib GUI backend, and automatically select the proper threading model to prevent blocking. It also sets matplotlib in interactive mode and modifies `%run` slightly, so that any matplotlib-based script can be executed using `%run` and the final `show()` command does not block the interactive shell.

The `-pylab` option must be given first in order for IPython to configure its threading mode. However, you can still issue other options afterwards. This allows you to have a matplotlib-based environment customized with additional modules using the standard IPython profile mechanism: "`ipython -pylab -p myprofile`" will load the profile defined in `ipythonrc-myprofile` after configuring matplotlib.

CHAPTER 4

# Introduction to numerix arrays

To be written...

CHAPTER 5

# Introduction to plotting with matplotlib / pylab

## 5.1. A bird's eye view

matplotlib is a library for making 2D plots of arrays in python.[1] Although it has its origins in emulating the Matlab graphics commands, it does not require matlab, and has a pure, object oriented API. Although matplotlib is written primarily in python, it makes heavy use of Numeric/numarray and other extension code to provide good performance even for large arrays. matplotlib is designed with the philosophy that you should be able to create simple plots with just a few commands, or just one! If you want to see a histogram of your data, you shouldn't need to instantiate objects, call methods, set properties, and so on; it should just work.

The matplotlib code is divided into three parts: the *pylab interface* is the set of functions provided by the `pylab` module which allow the user to create plots with code quite similar to matlab figure generating code. The matplotlib frontend or *matplotlib API* is the set of classes that do the heavy lifting, creating and managing figures, text, lines, plots and so on. This is an abstract interface that knowns nothing about output formats. The *backends* are device dependent drawing devices, aka renderers, that transform the frontend representation to hardcopy or a display device. Example backends: PS creates postscript hardcopy, SVG creates scalar vector graphics hardcopy, Agg creates PNG output using the high quality antigrain library that ships with matplotlib, GTK embeds matplotlib in a GTK application, GTKAgg uses the antigrain[2] renderer to create a figure and embed it a GTK application, and so on for WX, Tkinter, FLTK, ....

For years, I used to use matlab exclusively for data analysis and visualization. matlab excels at making nice looking plots easy. When I began working with EEG data, I found that I needed to write applications to interact with my data, and developed and EEG analysis application in matlab. As the application grew in complexity, interacting with databases, http servers, manipulating complex data structures, I began to strain against the limitations of matlab as a programming language, and decided to start over in python. python more than makes up for all of matlab's deficiencies as a programming language, but I was having difficulty finding a 2D plotting package – for 3D VTK, which is discussed at length below more than exceeds all of my needs.

When I went searching for a python plotting package, I had several requirements:

- Plots should look great - publication quality. One important requirement for me is that the text looks good (antialiased, etc)
- Postscript output for inclusion with LaTeX documents and publication quality printing
- Embeddable in a graphical user interface for application development
- The code should be mostly python so itis easy to understand and extend – users become developers!
- Making plots should be easy – just a few lines of code for simple graphs

Finding no package that suited me just right, I did what any self-respecting python programmer would do: rolled up my sleeves and dived in. Not having any real experience with computer graphics, I decided to emulate matlab's plotting capabilities because that is something matlab does very well. This had the added advantage that many people have a lot of matlab experience, and thus they can quickly get up to steam plotting in python. From a developer's perspective, having a fixed user interface (the pylab interface) has been very useful, because the guts of the code base can be redesigned without affecting user code.

Without further ado, let's create our first figure. This example uses the matplotlib object oriented API. Most users use the pylab interface, which will be discussed next and makes it easier to make plots because a lot of the tedius work of creating and managing figures and figure windows is done for you behind the hood. But since the real core of the library is the object oriented API, I think it is a good place to start. If you are developing a graphical user interface or making plots on a web server, you probably want maximal control with no magic going on behind the scenes – this is where the matplotlib API should be used. If you are just trying to make a figure for inclusion in a paper or if your working interactively from the python shell, you'll probably be happy with the pylab interface.

[1] This short guide is not meant as a complete guide or tutorial. There is a more comprehensive user's guide and tutorial on the matplotlib web-site at http://matplotlib.sf.net.

[2] http://antigrain.com

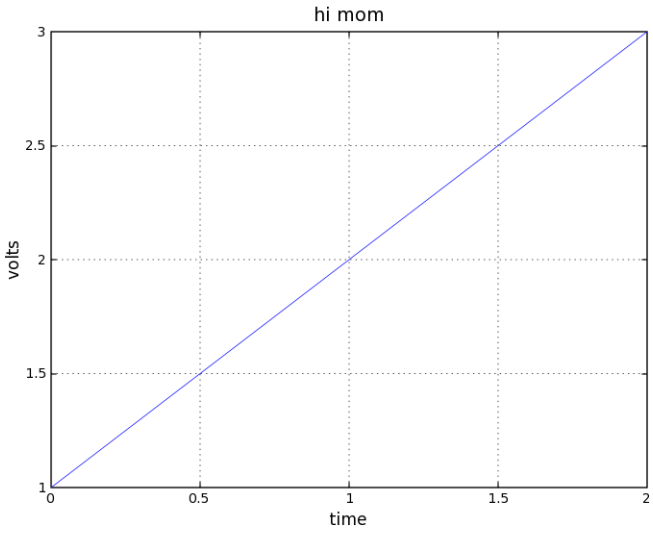



FIGURE 5.1.1. A simple plot generated by the antigrain (Agg) backend .

LISTING 5.1

```
"""
A pure object oriented example using the agg backend
"""
# import the matplotlib backend you want to use and the Figure class
from matplotlib.backends.backend_agg import FigureCanvasAgg as FigureCanvas
from matplotlib.figure import Figure

# the figure is the center of the action, and the canvas is a backend
# dependent container to hold the figure and make backend specific calls
fig = Figure()
canvas = FigureCanvas(fig)

# you can add multiple subplots and axes
ax = fig.add_subplot(111)

# the simplest plot!
ax.plot([1,2,3])

# you can decorate your plot with text and grids
ax.set_title('hi mom')
ax.grid(True)
ax.set_xlabel('time')
ax.set_ylabel('volts')

# and save it to hardcopy
fig.savefig('../fig/mpl_one_two_three.png')
```

### 5.2. A short pylab tutorial

Here is about the simplest code you can use to create a figure with matplotlib using the pylab interface. In this section, I'm assuming you are using ipython in the pylab mode – see Section3.8 for details.

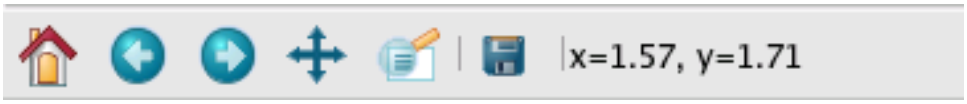



FIGURE 5.2.1. The matplotlib toolbar used to navigate around your figure

```
peds-pc311:~> pylab
Python 2.3.3 (#2, Apr 13 2004, 17:41:29)
Type "copyright", "credits" or "license" for more information.

IPython 0.6.12_cvs -- An enhanced Interactive Python.
?       -> Introduction to IPython's features.
%magic  -> Information about IPython's 'magic' % functions.
help    -> Python's own help system.
object? -> Details about 'object'. ?object also works, ?? prints more.

  Welcome to pylab, a matplotlib-based Python environment
    help(matplotlib) -> generic matplotlib information
    help(pylab)      -> matlab-compatible commands from matplotlib
    help(plotting)   -> plotting commands

In [1]: plot([1,2,3])
Out[1]: [<matplotlib.lines.Line2D instance at 0xb557a86c>]
```

If your settings are correct, a figure window should popup and you should be able to interact with it. That's a lot less typing than our initial example using the object oriented API in which you had to manually create the Figure, Axes and so on!

Try clicking on the navigation toolbar at the bottom of the figure – the toolbar is shown in Figure5.2.1. The first three buttons from left to right in Figure5.2.1 are *home*, *back* and *forward*. These byttons are are akin to the web browser buttons. They are used to navigate back and forth between previously defined views. They have no meaning unless you have already navigated somewhere else using the pan and zoom buttons as described below. This is analogous to trying to click `Back` on your web browser before visiting a new page –nothing happens. The home button always takes you to the first, default view of your data.

The next to button moving right is the pan/zoom button, which looks like a cross with arrows on the end (a *fleur*). The pan/zoom button button has two modes: pan and zoom (no surprise there, right?). Click this toolbar button to activate this mode; you should see "pan/zoom mode" show up in the status bar. Then put your mouse somewhere over an axes. To activate panning: press the left mouse button and hold it, dragging it to a new position. If you press `x` or `y` while panning, the motion will be contrained to the x or y axis, respectively . To activate zooming, press the right mouse button, dragging it to a new position. The x axis will be zoomed in proportionate to the rightward movement and zoomed out proportionate to the leftward movement. Ditto for the yaxis and up/down motions. The point under your mouse when you begin the zoom remains stationary, allowing you to zoom to an arbitrary point in the figure. You can use the modifier keys `x`, `y` or `CONTROL` to constrain the zoom to the x axes, the y axes, or aspect ratio preserve, respectively.

The next button moving right is the *zoom to rectangle button* which has a magnifying glass over a piece of paper. The button is striaghtforward and works in the standard way; when you click it, you should see that it is activated by looking for "Zoom to rect mode" in the status bar, and then you select the rectangular region you want to zoom in on.

The final button is the *save button*, which will save your figure in the current view. All of the *Agg backends know how to save the following image types: PNG, PS, EPS, SVG.

Let's make the same figure we made using the object oriented API above, ie Figure5.1.1, but this time using the pylab

LISTING 5.2

```
from pylab import *
plot([1,2,3])
title('hi mom')
grid(True)
xlabel('time')
```

```
ylabel('volts')
savefig('../fig/mpl_one_two_three.png')
show()
```

As you can see there is basically a direct translation between the OO interface and the pylab interface. When `plot` is called, the pylab interface makes a call to the function `gca()` ("get current axes") to get a reference to the current axes. `gca` in turn, makes a call to `gcf` ("get current figure") to get a reference to the current figure. `gcf`, finding that no figure has been created, creates the default figure using `figure()` and returns it. `gca` will then return the current axes of that figure if it exists, or create the default axes `subplot(111)` if it does not. The last line show is a GUI independent way of actually creating a figure window, and is not required for image backends such as postscript.

Thus a lot happens under the hood when you call plot, but for the most part you don't need to think about it – it just works. The important thing to understand is that the pylab interface has a state, and keeps track of the current figure and axes. All plotting commands target the current axes, and you can manipulate which ones are current

LISTING 5.3

```
from pylab import *

def f(t):
    s1 = cos(2*pi*t)
    e1 = exp(-t)
    return multiply(s1,e1)

t1 = arange(0.0, 5.0, 0.1)
t2 = arange(0.0, 5.0, 0.02)
t3 = arange(0.0, 2.0, 0.01)

# create and upper subplot and make it current
subplot(211)
l1, l2 = plot(t1, f(t1), 'bo', t2, f(t2), 'k--')
set(l1, markerfacecolor='g')
grid(True)
title('A tale of 2 subplots')
ylabel('Damped oscillation')

# create a lower subplot and make it current
subplot(212)
plot(t3, cos(2*pi*t3), 'r.')
grid(True)
xlabel('time (s)')
ylabel('Undamped')
savefig('../fig/mpl_subplot_demo')
show()
```

In addition to creating multiple subplots, this example contains a couple of new things. In the first plot command, the return value is stored as `l1, l2` and the `set` command is used to change a default line property.

```
l1, l2 = plot(t1, f(t1), 'bo', t2, f(t2), 'k--')
set(l1, markerfacecolor='g')
```

`l1` and `l2` are `matplotlib.lines.Line2D` instances and they are created by the `plot` command and added to the current axes. This is the typical mode of operation of the axes plot commands: they create a bunch of primitive objects (lines, polygons, text, images), add them to the axes, and return them. In this example, the line's `markerfacecolor` property is set with the `set` command. In the next section, we'll look into matplotlibs `set` and `get` introspection system and show how to use it to customize your lines, polygons, text instances and images.

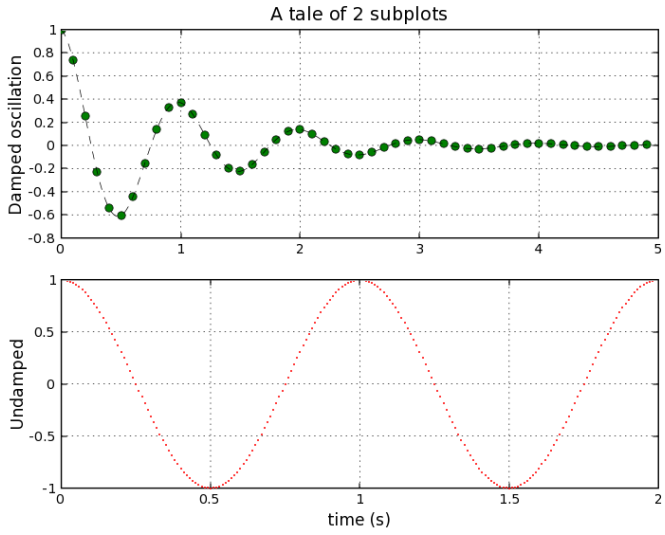



FIGURE 5.2.2. It's easy to create multiple axes and subplots.

### 5.3. Set and get introspection

Everything that goes into a matplotlib figure, including the `Figure` itself, are all objects dervied from a single base class `Artist`, and the pylab `set` and `get` commands provide a unified way to configure them. Let's create a simple plot of random circles, and use that to explore how `set` and get work. First the basic plot – we'll store the return value as lines. Note that `plot` always returns a *list* of lines; in the example above there were two lines `l1` and `l2`, and in the example below there is only a single element of the list lines. No matter: `set` and `get` will work on a single instance or a sequence of instances

```
In [2]: x = rand(20); y = rand(20)

In [3]: lines = plot(x,y,'o')

In [4]: type(lines)         # plot always returns a list
Out[4]: <type 'list'>

In [5]: len(lines)          # even if it is length 1
Out[5]: 1
```

The simple figure that was created, a scattering of blue circles at random locations, is shown in Figure5.3.1. To see a listing of the properties of the line, and what their current values are, call `get(lines)`

```
In [29]: get(lines)
    alpha = 1.0
    antialiased or aa = True
    clip_on = True
    color or c = blue
    figure = <matplotlib.figure.Figure instance at 0xb40e1cec>
    label =
    linestyle or ls = None
    linewidth or lw = 0.5
    marker = o
    markeredgecolor or mec = black
    markeredgewidth or mew = 0.5
    markerfacecolor or mfc = blue
```

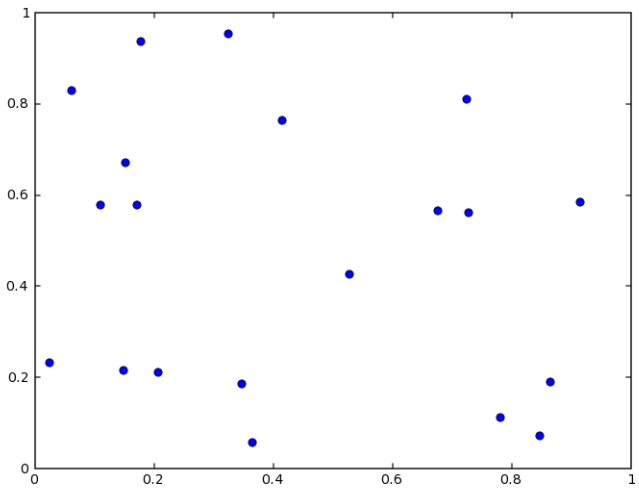



FIGURE 5.3.1. The default marker plot, before marker customization

```
    markersize or ms = 6.0
    transform = <Affine object at 0x8683c6c>
    visible = True
    xdata = [ 0.16952688  0.59729624  0.16829208  0.51311375  0.7227286
        ...0.45925692]...
    ydata = [ 0.86459035  0.25595992  0.01905832  0.24303582  0.74993261
        ...0.28751132]...
    zorder = 2
```

and to see the same listing of properties with information on legal values you can set them to, call `set(lines)`

```
In [37]: set(lines)
    alpha: float
    antialiased or aa: [True | False]
    clip_box: a matplotlib.transform.Bbox instance
    clip_on: [True | False]
    color or c: any matplotlib color - see help(colors)
    dashes: sequence of on/off ink in points
    data: (array xdata, array ydata)
    data_clipping: [True | False]
    figure: a matplotlib.figure.Figure instance
    label: any string
    linestyle or ls: [ '-' | '--' | '-.' | ':' | 'steps' | 'None' ]
    linewidth or lw: float value in points
    lod: [True | False]
    marker: [ '+' | ',' | '.' | '1' | '2' | '3' | '4' | '<' | '>' | 'D' | 'H' |
        ... '^' | '_' | 'd' | 'h' | 'o' | 'p' | 's' | 'v' | 'x' | '|' ]
    markeredgecolor or mec: any matplotlib color - see help(colors)
    markeredgewidth or mew: float value in points
    markerfacecolor or mfc: any matplotlib color - see help(colors)
    markersize or ms: float
    transform: a matplotlib.transform transformation instance
    visible: [True | False]
    xclip: (xmin, xmax)
```

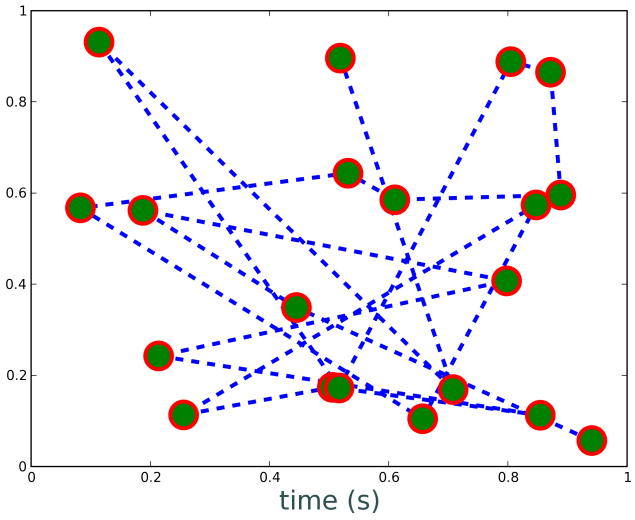



FIGURE 5.3.2. The default marker plot, before marker customization

```
xdata: array
yclip: (ymin, ymax)
ydata: array
zorder: any number
```

OK, we have a lot of options here. Let's change the marker properties, and add a linesytle

```
In [20]: set(lines, markerfacecolor='green', markeredgecolor='red',
   ....:  markersize=20, markeredgewidth=3,
   ....: linestyle='--', linewidth=3)
```

That's a lot of typing, but to great effect! The same data set now has quite a different appearance, which is shown in Figure5.3.2. Note in the long listing output of the set(lines) command above the markerfacecolor settable property is listed as

```
markerfacecolor or mfc: any matplotlib color - see help(colors)
```

The `markerfacecolor` has an alias `mfc` to save typing, and common colornames have abbreviations too, so the `set` command above could just as well be written

```
In [20]: set(lines, mfc='g', mec='r', ms=20, mew=3, ls='--', lw=3)
```

Another nice thing about matplotlib properties is that you can pass them in as keyword arguments to `plot` and they will have the same effect, eg, you can create the identical plot with

```
In [6]: plot(x, y, 'o', mfc='g', mec='r', ms=20, mew=3, ls='--', lw=3)
Out[6]: [<matplotlib.lines.Line2D instance at 0xb40db42c>]
```

As noted above, `set` and `get` work on any `Artist`, so you can configure your axes or text instances this way. Eg, `xlabel` returns a `matplotlib.text.Text` instance

```
In [8]: t = xlabel('time (s)')

In [9]: set(t)
    alpha: float
    backgroundcolor: any matplotlib color - see help(colors)
    bbox: rectangle prop dict plus key 'pad' which is a pad in points
    clip_box: a matplotlib.transform.Bbox instance
    clip_on: [True | False]
    color: any matplotlib color - see help(colors)
    family: [ 'serif' | 'sans-serif' | 'cursive' | 'fantasy' | 'monospace' ]
```

```
figure: a matplotlib.figure.Figure instance
fontproperties: a matplotlib.font_manager.FontProperties instance
horizontalalignment or ha: [ 'center' | 'right' | 'left' ]
label: any string
lod: [True | False]
multialignment: ['left' | 'right' | 'center' ]
name or fontname: string eg, ['Sans' | 'Courier' | 'Helvetica' ...]
position: (x,y)
rotation: [ angle in degrees 'vertical' | 'horizontal'
size or fontsize: [ size in points | relative size eg 'smaller', 'x-large'
    ...]    style or fontstyle: [ 'normal' | 'italic' | 'oblique']
text: string
transform: a matplotlib.transform transformation instance
variant: [ 'normal' | 'small-caps' ]
verticalalignment or va: [ 'center' | 'top' | 'bottom' ]
visible: [True | False]
weight or fontweight: [ 'normal' | 'bold' | 'heavy' | 'light' | 'ultrabold'
    ... | 'ultralight']
x: float
y: float
zorder: any number
```

So you have a lot of possibilities to customize your text! The most common things people what to do are change the font size and color; the results of this command on the xlabel are shown in Figure5.3.2.

```
In [25]: set(t, fontsize=20, color='darkslategray')
```

### 5.4. A common interface to Numeric and numarray

Currently the python computing community is in a state of having too many array pacakges, none of which satisfy everyone's needs. Although Numeric and numarray both provide the same set of core functions, they are organized differently, and matplotlib provides a compatibility later so you can use either one in your matplotlib scripts without having to change your code.

Several numarray/Numeric developers are codevelopers of matplotlib, giving matplotlib full Numeric and numarray compatibility, thanks in large part to Todd Miller's `matplotlib.numerix` module and the numarray compatibility layer for extension code. This allows you to choose between Numeric or numarray at the prompt or in a config file. Thus when you do

```
# import matplotlib and all the numerix functions
from pylab import *
```

you'll not only get all the matplotlib pylab interface commands, but most of the Numeric or numarray package as well (depending on your `numerix` setting). All of the array creation and manipulation functions are imported, such as `array`, `arange`, `take`, `where`, etc, as are the external module functions which reside in `mlab, fft` and `linear_algebra`.

Even if you don't want to import all of the numerix symbols from the pytlab interface, to make your matplotlib scripts as portable as possible with respect to your choice of array packages, it is advised not to explicitly import Numeric or numarray. Rather, you should use `matplotlib.numerix` where possible, either by using the functions imported by `pylab`, or by explicitly importing the `numerix` module, as in

```
# create a numerix namespace
import matplotlib.numerix as n
from matplotlib.numerix.mlab import mean
x = n.arange(100)
y = n.take(x, range(10,20))
print mean(y)
```

For the remainder of this manual, the term `numerix` is used to mean either the Numeric or numarray package. To select numarray or Numeric from the prompt, run your matplotlib script with

```
> python myscript.py --numarray  # use numarray
> python myscript.py --Numeric   # use Numeric
```

Typically, however, users will choose one or the other and make this setting in their rc file using either `numerix : Numeric` or `numerix : numarray`.

### 5.5. Customizing the default behavior with the rc file

matplotlib is designed to work in a variety of settings: some people use it in "batch mode" on a web server to create images they never look at. Others use graphical user interfaces (GUIs) to interact with their plots. Thus you must customize matplotlib to work like you want it to with the customization file `.matplotlibrc`, in which you can set whether you want to just create images or use a GUI (the backend setting), and whether you want to work interactively from the shell (the interactive setting). Almost all of the matplotlib settings and figure properties can be customized with this file, which is installed with the rest of the matplotlib data (fonts, icons, etc) into a directory determined by distutils. Before compiling matplotlib, it resides in the same dir as `setup.py` and will be copied into your install path. Typical locations for this file are

```
C:\Python23\share\matplotlib\.matplotlibrc # windows /usr/share/matplotlib/.matplotlibrc  # linux
```

By default, the installer will overwrite the existing file in the install path, so if you want to preserve your's, please move it to your `HOME` dir and set the environment variable if necessary. In the rc file, you can set your backend , your numerix setting , whether you'll be working interactively and default values for most of the figure properties.

In the RC file, blank lines, or lines starting with a comment symbol, are ignored, as are trailing comments. Other lines must have the format

```
key : val # optional comment
```

where *key* is some property like `backend`, `lines.linewidth`, or `figure.figsize` and *val* is the value of that property. Example entries for these properties are

```
# this is a comment and is ignored
backend         : GTKAgg    # the default backend
lines.linewidth : 0.5       # line width in points
figure.figsize  : 8, 6      # figure size in inches
```

A complete sample rc file is included with the matplotlib distribution and available online.[3]

### 5.6. A quick tour of plot types

### 5.7. Images

Matplotlib has support for plotting images with imshow and figimage. In imshow, the image data is scaled to fit into the current axes, and many different interpolation schemes are supported to do the resampling, and in figimage, the image data are transferred as a raw pixel dump to the figure canvas without resampling. You can add colorbars, set the default colormaps, and change the interpolation schemes quite easily.

```
In [15]: x = arange(100.0); x.shape = 10,10

In [16]: im = imshow(x, interpolation='nearest')

In [17]: colorbar()
Out[17]: <matplotlib.axes.Axes instance at 0xb455496c>
```

which creates the image shown in Figure 5.7.1. You can interactively update the default colormap and change the interpolation scheme, which creates the image show in Figure 5.7.2.

```
In [18]: im.set_interpolation('bilinear')

In [19]: hot()
```

There is a lot more you can do with images: you can set the data extent so that you can overlay contours or other plots, you can plot multiple images to the same axes with different colors and transparency values, you can load images with PIL or `imread` and plot them in matplotlib, you can create montages of with `figimage` placed around the figure window at different offsets, you can plot grayscale, rgb or rgba data, and so on. Consult the *Matplotlib User's Guide* and the `examples` subdirectory in the matplotlib source distribution for more information. We'll clost off with a simple example of reading in a PNG and displaying it

[3] http://matplotlib.sourceforge.net/.matplotlibrc

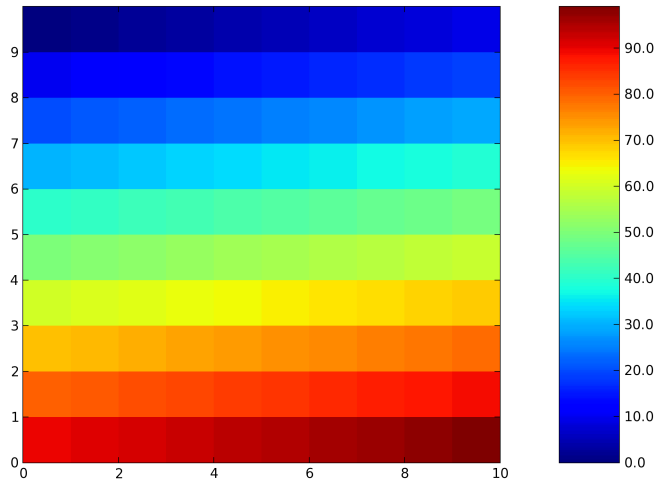



FIGURE 5.7.1. A simple image plot of a 2D matrix, using nearest neighbor interpolation and the `jet` colormap.

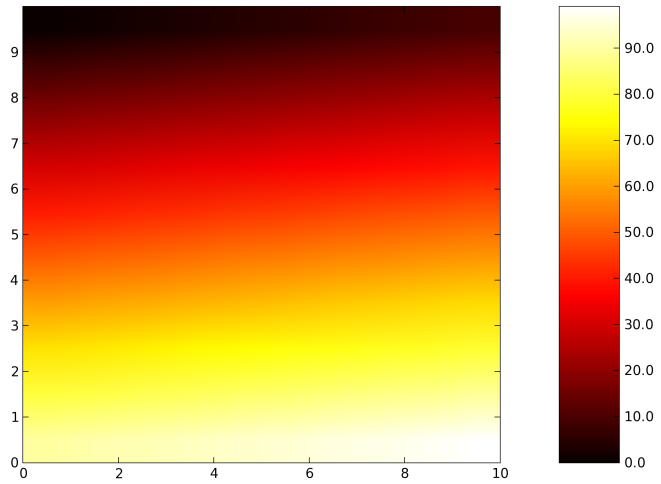



FIGURE 5.7.2. The same image data, rendered with the hot colormap and bilinear interpolation. matplotlib has 14 colormaps built-in, and you can define your own with relative ease, and there are 16 interpolation methods.

```
In [35]: im = imread('../data/ratner.png')

In [36]: imshow(im)
Out[36]: <matplotlib.image.AxesImage instance at 0xb3ffba2c>

In [37]: axis('off')
```

FIGURE 5.7.3. Displaying image data from your camera in matplotlib

**5.8. Customizing text and mathematical expressions**

**5.9. Event handling: Tracking the mouse and keyboard**

CHAPTER 6

# A tour of SciPy

Chapter contributed by Travis E. Oliphant

## 6.1. Introduction

SciPy is a collection of mathematical algorithms and convenience functions built on the Numeric extension for Python. It adds significant power to the interactive Python session by exposing the user to high-level commands and classes for the manipulation and visualization of data. With SciPy, an interactive Python session becomes a data-processing and system-prototyping environment rivaling sytems such as Matlab, IDL, Octave, R-Lab, and SciLab.

The additional power of using SciPy within Python, however, is that a powerful programming language is also available for use in developing sophisticated programs and specialized applications. Scientific applications written in SciPy benefit from the development of additional modules in numerous niche's of the software landscape by developers across the world. Everything from parallel programming to web and data-base subroutines and classes have been made available to the Python programmer. All of this power is available in addition to the mathematical libraries in SciPy.

This document provides a tutorial for the first-time user of SciPy to help get started with some of the features available in this powerful package. It is assumed that the user has already installed the package. Some general Python facility is also assumed such as could be acquired by working through the Tutorial in the Python distribution. For further introductory help the user is directed to the Numeric documentation. Throughout this tutorial it is assumed that the user has imported all of the names defined in the SciPy namespace using the command

```
>>> from scipy import *
```

**6.1.1. General Help.** Python provides the facility of documentation strings. The functions and classes available in SciPy use this method for on-line documentation. There are two methods for reading these messages and getting help. Python provides the command **help** in the pydoc module. Entering this command with no arguments (i.e. >>> help ) launches an interactive help session that allows searching through the keywords and modules available to all of Python. Running the command help with an object as the argument displays the calling signature, and the documentation string of the object.

The pydoc method of help is sophisticated but uses a pager to display the text. Sometimes this can interfere with the terminal you are running the interactive session within. A scipy-specific help system is also available under the command scipy.info. The signature and documentation string for the object passed to the help command are printed to standard output (or to a writeable object passed as the third argument). The second keyword argument of "scipy.info" defines the maximum width of the line for printing. If a module is passed as the argument to help than a list of the functions and classes defined in that module is printed. For example:

```
>>> info(optimize.fmin)
 fmin(func, x0, args=(), xtol=0.0001, ftol=0.0001, maxiter=None, maxfun=None,
      full_output=0, printmessg=1)

Minimize a function using the simplex algorithm.

Description:

  Uses a Nelder-Mead simplex algorithm to find the minimum of function
  of one or more variables.

Inputs:

  func -- the Python function or method to be minimized.
```

```
  x0 -- the initial guess.
  args -- extra arguments for func.
  xtol -- relative tolerance

Outputs: (xopt, {fopt, warnflag})

  xopt -- minimizer of function

  fopt -- value of function at minimum: fopt = func(xopt)
  warnflag -- Integer warning flag:
              1 : 'Maximum number of function evaluations.'
              2 : 'Maximum number of iterations.'

Additional Inputs:

  xtol -- acceptable relative error in xopt for convergence.
  ftol -- acceptable relative error in func(xopt) for convergence.
  maxiter -- the maximum number of iterations to perform.
  maxfun -- the maximum number of function evaluations.
  full_output -- non-zero if fval and warnflag outputs are desired.
  printmessg -- non-zero to print convergence messages.
```

Another useful command is **source.** When given a function written in Python as an argument, it prints out a listing of the source code for that function. This can be helpful in learning about an algorithm or understanding exactly what a function is doing with its arguments. Also don't forget about the Python command `dir` which can be used to look at the namespace of a module or package.

**6.1.2. SciPy Organization.** SciPy is organized into subpackages covering different scientific computing domains. Some common functions which several subpackages rely on live under the `scipy_base` package which is installed at the same directory level as the scipy package itself and could be installed separately. This allows for the possibility of separately distributing the subpackages of scipy as long as scipy_base package is provided as well.

Two other packages are installed at the higher-level: scipy_distutils and weave. These two packages while distributed with main scipy package could see use independently of scipy and so are treated as separate packages and described elsewhere.

The remaining subpackages are summarized in the following table (a * denotes an optional sub-package that requires additional libraries to function or is not available on all platforms).

| Subpackage | Description |
|---|---|
| cluster | Clustering algorithms |
| cow | Cluster of Workstations code for parallel programming |
| fftpack | FFT based on fftpack – default |
| fftw* | FFT based on fftw — requires FFTW libraries (is this still needed?) |
| ga | Genetic algorithms |
| gplt* | Plotting — requires gnuplot |
| integrate | Integration |
| interpolate | Interpolation |
| io | Input and Output |
| linalg | Linear algebra |
| optimize | Optimization and root-finding routines |
| plt* | Plotting — requires wxPython |
| signal | Signal processing |
| special | Special functions |
| stats | Statistical distributions and functions |
| xplt | Plotting with gist |

Because of their ubiquitousness, some of the functions in these subpackages are also made available in the scipy namespace to ease their use in interactive sessions and programs. In addition, many convenience functions

are located in the scipy_base package and the in the top-level of the scipy package. Before looking at the sub-packages individually, we will first look at some of these common functions.

### 6.2. Basic functions in scipy_base and top-level scipy

**6.2.1. Interaction with Numeric.** To begin with, all of the Numeric functions have been subsumed into the scipy namespace so that all of those functions are available without additionally importing Numeric. In addition, the universal functions (addition, subtraction, division) have been altered to not raise exceptions if floating-point errors are encountered[1], instead NaN's and Inf's are returned in the arrays. To assist in detection of these events new universal functions (isnan, isfinite, isinf) have been added.

In addition, the comparision operators have been changed to allow comparisons and logical operations of complex numbers (only the real part is compared). Also, with the new universal functions in SciPy, the logical operations (except logical_XXX functions) all return arrays of unsigned bytes (8-bits per element instead of the old 32-bits, or even 64-bits) per element[2].

In an effort to get a consistency for default arguments, some of the default arguments have changed from Numeric. The idea is for you to use scipy as a base package instead of Numeric anyway.

Finally, some of the basic functions like log, sqrt, and inverse trig functions have been modified to return complex numbers instead of NaN's where appropriate (*i.e.* `scipy.sqrt(-1)` returns `1j`).

**6.2.2. Alter numeric.** With the command **scipy.alter_numeric()** you can now use index and mask arrays inside brackets and the coercion rules of Numeric will be changed so that Python scalars will not be used to determine the type of the output of an expression.

**6.2.3. Scipy_base routines.** The purpose of scipy_base is to collect general-purpose routines that the other sub-packages can use and to provide a simple replacement for Numeric. Anytime you might think to import Numeric, you can import scipy_base instead and remove yourself from direct dependence on Numeric. These routines are divided into several files for organizational purposes, but they are all available under the scipy_base namespace (and the scipy namespace). There are routines for type handling and type checking, shape and matrix manipulation, polynomial processing, and other useful functions. Rather than giving a detailed description of each of these functions (which is available using the **help**, **info** and **source** commands), this tutorial will discuss some of the more useful commands which require a little introduction to use to their full potential.

6.2.3.1. *Type handling.* Note the difference between **iscomplex** (**isreal**) and **iscomplexobj** (**isrealobj**). The former command is array based and returns byte arrays of ones and zeros providing the result of the element-wise test. The latter command is object based and returns a scalar describing the result of the test on the entire object.

Often it is required to get just the real and/or imaginary part of a complex number. While complex numbers and arrays have attributes that return those values, if one is not sure whether or not the object will be complex-valued, it is better to use the functional forms **real** and **imag**. These functions succeed for anything that can be turned into a Numeric array. Consider also the function **real_if_close** which transforms a complex-valued number with tiny imaginary part into a real number.

Occasionally the need to check whether or not a number is a scalar (Python (long)int, Python float, Python complex, or rank-0 array) occurs in coding. This functionality is provided in the convenient function **isscalar** which returns a 1 or a 0.

Finally, ensuring that objects are a certain Numeric type occurs often enough that it has been given a convenient interface in SciPy through the use of the **cast** dictionary. The dictionary is keyed by the type it is desired to cast to and the dictionary stores functions to perform the casting. Thus, `>>> a = cast['f'](d)` returns an array of float32 from d. This function is also useful as an easy way to get a scalar of a certain type: `>>> fpi = cast['f'](pi)` although this should not be needed if you use alter_numeric().

6.2.3.2. *Index Tricks.* Thre are some class instances that make special use of the slicing functionality to provide efficient means for array construction. This part will discuss the operation of **mgrid**, **ogrid**, **r_**, and **c_** for quickly constructing arrays.

One familiar with Matlab may complain that it is difficult to construct arrays from the interactive session with Python. Suppose, for example that one wants to construct an array that begins with 3 followed by 5 zeros and then contains 10 numbers spanning the range -1 to 1 (inclusive on both ends). Before SciPy, you would need to enter something like the following

[1]These changes are all made in a new module (fastumath) that is part of the scipy_base package. The old functionality is still available in umath (part of Numeric) if you need it (note: importing umath or fastumath resets the behavior of the infix operators to use the umath or fastumath ufuncs respectively).

[2]Be careful when treating logical expressions as integers as the 8-bit integers may silently overflow at 256.

```
>>> concatenate(([3],[0]*5,arange(-1,1.002,2/9.0)).
```

With the **r_** command one can enter this as

```
>>> r_[3,[0]*5,-1:1:10j]
```

which can ease typing and make for more readable code. Notice how objects are concatenated, and the slicing syntax is (ab)used to construct ranges. The other term that deserves a little explanation is the use of the complex number 10j as the step size in the slicing syntax. This non-standard use allows the number to be interpreted as the number of points to produce in the range rather than as a step size (note we would have used the long integer notation, 10L, but this notation may go away in Python as the integers become unified). This non-standard usage may be unsightly to some, but it gives the user the ability to quickly construct complicated vectors in a very readable fashion. When the number of points is specified in this way, the end-point is inclusive.

The "r" stands for row concatenation because if the objects between commas are 2 dimensional arrays, they are stacked by rows (and thus must have commensurate columns). There is an equivalent command **c_** that stacks 2d arrays by columns but works identically to **r_** for 1d arrays.

Another very useful class instance which makes use of extended slicing notation is the function **mgrid**. In the simplest case, this function can be used to construct 1d ranges as a convenient substitute for arange. It also allows the use of complex-numbers in the step-size to indicate the number of points to place between the (inclusive) end-points. The real purpose of this function however is to produce N, N-d arrays which provide coordinate arrays for an N-dimensional volume. The easiest way to understand this is with an example of its usage:

```
>>> mgrid[0:5,0:5]
array([[[0, 0, 0, 0, 0],
        [1, 1, 1, 1, 1],
        [2, 2, 2, 2, 2],
        [3, 3, 3, 3, 3],
        [4, 4, 4, 4, 4]],
       [[0, 1, 2, 3, 4],
        [0, 1, 2, 3, 4],
        [0, 1, 2, 3, 4],
        [0, 1, 2, 3, 4],
        [0, 1, 2, 3, 4]]])
>>> mgrid[0:5:4j,0:5:4j]
array([[[ 0.    ,  0.    ,  0.    ,  0.    ],
        [ 1.6667,  1.6667,  1.6667,  1.6667],
        [ 3.3333,  3.3333,  3.3333,  3.3333],
        [ 5.    ,  5.    ,  5.    ,  5.    ]],
       [[ 0.    ,  1.6667,  3.3333,  5.    ],
        [ 0.    ,  1.6667,  3.3333,  5.    ],
        [ 0.    ,  1.6667,  3.3333,  5.    ],
        [ 0.    ,  1.6667,  3.3333,  5.    ]]])
```

Having meshed arrays like this is sometimes very useful. However, it is not always needed just to evaluate some N-dimensional function over a grid due to the array-broadcasting rules of Numeric and SciPy. If this is the only purpose for generating a meshgrid, you should instead use the function **ogrid** which generates an "open" grid using NewAxis judiciously to create N, N-d arrays where only one-dimension in each array has length greater than 1. This will save memory and create the same result if the only purpose for the meshgrid is to generate sample points for evaluation of an N-d function.

6.2.3.3. *Shape manipulation.* In this category of functions are routines for squeezing out length-one dimensions from N-dimensional arrays, ensuring that an array is at least 1-, 2-, or 3-dimensional, and stacking (concatenating) arrays by rows, columns, and "pages" (in the third dimension). Routines for splitting arrays (roughly the opposite of stacking arrays) are also available.

6.2.3.4. *Matrix manipulations.* These are functions specifically suited for 2-dimensional arrays that were part of MLab in the Numeric distribution, but have been placed in scipy_base for completeness so that users are not importing Numeric.

6.2.3.5. *Polynomials.* There are two (interchangeable) ways to deal with 1-d polynomials in SciPy. The first is to use the **poly1d** class in **scipy_base**. This class accepts coefficients or polynomial roots to initialize a polynomial. The polynomial object can then be manipulated in algebraic expressions, integrated, differentiated, and evaluated. It even prints like a polynomial:

```
>>> p = poly1d([3,4,5])
>>> print p
```

```
   2
3 x + 4 x + 5
>>> print p*p
   4      3      2
9 x + 24 x + 46 x + 40 x + 25
>>> print p.integ(k=6)
 3     2
x + 2 x + 5 x + 6
>>> print p.deriv()
6 x + 4
>>> p([4,5])
array([ 69, 100])
```

The other way to handle polynomials is as an array of coefficients with the first element of the array giving the coefficient of the highest power. There are explicit functions to add, subtract, multiply, divide, integrate, differentiate, and evaluate polynomials represented as sequences of coefficients.

6.2.3.6. *Vectorizing functions (vectorize).* One of the features that SciPy provides is a class **vectorize** to convert an ordinary Python function which accepts scalars and returns scalars into a "vectorized-function" with the same broadcasting rules as other Numeric functions (*i.e.* the Universal functions, or ufuncs). For example, suppose you have a Python function named **addsubtract** defined as:

```
>>> def addsubtract(a,b):
    if a > b:
        return a - b
    else:
        return a + b
```

which defines a function of two scalar variables and returns a scalar result. The class vectorize can be used to "vectorize" this function so that

```
>>> vec_addsubtract = vectorize(addsubtract)
```

returns a function which takes array arguments and returns an array result:

```
>>> vec_addsubtract([0,3,6,9],[1,3,5,7])
array([1, 6, 1, 2])
```

This particular function could have been written in vector form without the use of **vectorize**. But, what if the function you have written is the result of some optimization or integration routine. Such functions can likely only be vectorized using `vectorize.`

6.2.3.7. *Other useful functions.* There are several other functions in the scipy_base package including most of the other functions that are also in MLab that comes with the Numeric package. The reason for duplicating these functions is to allow SciPy to potentially alter their original interface and make it easier for users to know how to get access to functions `>>> from scipy import *.`

New functions which should be mentioned are **mod(x,y)** which can replace **x%y** when it is desired that the result take the sign of **y** instead of **x**. Also included is **fix** which always rounds to the nearest integer towards zero. For doing phase processing, the functions **angle,** and **unwrap** are also useful. Also, the **linspace** and **logspace** functions return equally spaced samples in a linear or log scale. Finally, mention should be made of the new function **select** which extends the functionality of **where** to include multiple conditions and multiple choices. The calling convention is `select(condlist,choicelist,default=0).` **Select** is a vectorized form of the multiple if-statement. It allows rapid construction of a function which returns an array of results based on a list of conditions. Each element of the return array is taken from the array in a `choicelist` corresponding to the first condition in `condlist` that is true. For example

```
>>> x = r_[-2:3]
>>> x
array([-2, -1,  0,  1,  2])
>>> select([x > 3, x >= 0],[0,x+2])
array([0, 0, 2, 3, 4])
```

**6.2.4. Common functions.** Some functions depend on sub-packages of SciPy but should be available from the top-level of SciPy due to their common use. These are functions that might have been placed in scipy_base except for their dependence on other sub-packages of SciPy. For example the **factorial** and **comb** functions compute $n!$ and $n!/k!(n-k)!$ using either exact integer arithmetic (thanks to Python's Long integer object), or by using floating-point precision and the gamma function. The functions **rand** and **randn** are used

so often that they warranted a place at the top level. There are convenience functions for the interactive use: **disp** (similar to print), and **who** (returns a list of defined variables and memory consumption–upper bounded). Another function returns a common image used in image processing: **lena.**

Finally, two functions are provided that are useful for approximating derivatives of functions using discrete-differences. The function **central_diff_weights** returns weighting coefficients for an equally-spaced $N$-point approximation to the derivative of order $o$. These weights must be multiplied by the function corresponding to these points and the results added to obtain the derivative approximation. This function is intended for use when only samples of the function are avaiable. When the function is an object that can be handed to a routine and evaluated, the function **derivative** can be used to automatically evaluate the object at the correct points to obtain an N-point approximation to the $o^{\text{th}}$-derivative at a given point.

## 6.3. Special functions (special)

The main feature of the **special** package is the definition of numerous special functions of mathematical physics. Available functions include airy, elliptic, bessel, gamma, beta, hypergeometric, parabolic cylinder, mathieu, spheroidal wave, struve, and kelvin. There are also some low-level stats functions that are not intended for general use as an easier interface to these functions is provided by the `stats` module. Most of these functions can take array arguments and return array results following the same broadcasting rules as other math functions in Numerical Python. Many of these functions also accept complex-numbers as input. For a complete list of the available functions with a one-line description type `>>>info(special).` Each function also has it's own documentation accessible using help. If you don't see a function you need, consider writing it and contributing it to the library. You can write the function in either C, Fortran, or Python. Look in the source code of the library for examples of each of these kind of functions.

## 6.4. Integration (integrate)

The **integrate** sub-package provides several integration techniques including an ordinary differential equation integrator. An overview of the module is provided by the help command:

```
>>> help(integrate)
Methods for Integrating Functions

  odeint        -- Integrate ordinary differential equations.
  quad          -- General purpose integration.
  dblquad       -- General purpose double integration.
  tplquad       -- General purpose triple integration.
  gauss_quad    -- Integrate func(x) using Gaussian quadrature of order n.
  gauss_quadtol -- Integrate with given tolerance using Gaussian quadrature.

  See the orthogonal module (integrate.orthogonal) for Gaussian
     quadrature roots and weights.
```

**6.4.1. General integration (integrate.quad).** The function **quad** is provided to integrate a function of one variable between two points. The points can be $\pm\infty$ ($\pm$**integrate.inf**) to indicate infinite limits. For example, suppose you wish to integrate a bessel function `jv(2.5,x)` along the interval $[0, 4.5]$.

$$I = \int_0^{4.5} J_{2.5}(x)\, dx.$$

This could be computed using **quad:**

```
>>> result = integrate.quad(lambda x: special.jv(2.5,x), 0, 4.5)
>>> print result
(1.1178179380783249, 7.8663172481899801e-09)

>>> I = sqrt(2/pi)*(18.0/27*sqrt(2)*cos(4.5)-4.0/27*sqrt(2)*sin(4.5)+
    sqrt(2*pi)*special.fresnl(3/sqrt(pi))[0])
>>> print I
1.117817938088701

>>> print abs(result[0]-I)
1.03761443881e-11
```

The first argument to quad is a "callable" Python object (*i.e* a function, method, or class instance). Notice the use of a lambda-function in this case as the argument. The next two arguments are the limits of integration. The return value is a tuple, with the first element holding the estimated value of the integral and the second element holding an upper bound on the error. Notice, that in this case, the true value of this integral is

$$I = \sqrt{\frac{2}{\pi}}\left(\frac{18}{27}\sqrt{2}\cos\left(4.5\right) - \frac{4}{27}\sqrt{2}\sin\left(4.5\right) + \sqrt{2\pi}\mathrm{Si}\left(\frac{3}{\sqrt{\pi}}\right)\right),$$

where

$$\mathrm{Si}\left(x\right) = \int_0^x \sin\left(\frac{\pi}{2}t^2\right)\, dt.$$

is the Fresnel sine integral. Note that the numerically-computed integral is within $1.04 \times 10^{-11}$ of the exact result — well below the reported error bound.

Infinite inputs are also allowed in **quad** by using ±**integrate.inf** (or **inf)** as one of the arguments. For example, suppose that a numerical value for the exponential integral:

$$E_n\left(x\right) = \int_1^\infty \frac{e^{-xt}}{t^n}\, dt.$$

is desired (and the fact that this integral can be computed as `special.expn(n,x)` is forgotten). The functionality of the function **special.expn** can be replicated by defining a new function **vec_expint** based on the routine **quad:**

```
>>> from integrate import quad, Inf
>>> def integrand(t,n,x):
        return exp(-x*t) / t**n

>>> def expint(n,x):
        return quad(integrand, 1, Inf, args=(n, x))[0]

>>> vec_expint = vectorize(expint)

>>> vec_expint(3,arange(1.0,4.0,0.5))
array([ 0.1097,  0.0567,  0.0301,  0.0163,  0.0089,  0.0049])
>>> special.expn(3,arange(1.0,4.0,0.5))
array([ 0.1097,  0.0567,  0.0301,  0.0163,  0.0089,  0.0049])
```

The function which is integrated can even use the quad argument (though the error bound may underestimate the error due to possible numerical error in the integrand from the use of **quad**). The integral in this case is

$$I_n = \int_0^\infty \int_1^\infty \frac{e^{-xt}}{t^n}\, dt\, dx = \frac{1}{n}.$$

```
>>> result = quad(lambda x: expint(3, x), 0, Inf)
>>> print result
(0.33333333324560266, 2.8548934485373678e-09)

>>> I3 = 1.0/3.0
>>> print I3
0.333333333333

>>> print I3 - result[0]
8.77306560731e-11
```

This last example shows that multiple integration can be handled using repeated calls to **quad.** The mechanics of this for double and triple integration have been wrapped up into the functions **dblquad** and **tplquad.** The function, **dblquad** performs double integration. Use the help function to be sure that the arguments are defined in the correct order. In addition, the limits on all inner integrals are actually functions which can be constant functions. An example of using double integration to compute several values of $I_n$ is shown below:

```
>>> from __future__ import nested_scopes
>>> from integrate import quad, dblquad, Inf
>>> def I(n):
    return dblquad(lambda t, x: exp(-x*t)/t**n, 0, Inf, lambda x: 1, lambda x: Inf)
```

```
>>> print I(4)
(0.25000000000435768, 1.0518245707751597e-09)
>>> print I(3)
(0.33333333325010883, 2.8604069919261191e-09)
>>> print I(2)
(0.49999999999857514, 1.8855523253868967e-09)
```

**6.4.2. Gaussian quadrature (integrate.gauss_quadtol).** A few functions are also provided in order to perform simple Gaussian quadrature over a fixed interval. The first is **fixed_quad** which performs fixed-order Gaussian quadrature. The second function is **quadrature** which performs Gaussian quadrature of multiple orders until the difference in the integral estimate is beneath some tolerance supplied by the user. These functions both use the module **special.orthogonal** which can calculate the roots and quadrature weights of a large variety of orthogonal polynomials (the polynomials themselves are available as special functions returning instances of the polynomial class — e.g. `special.legendre`).

**6.4.3. Integrating using samples.** There are three functions for computing integrals given only samples: **trapz**, **simps**, and **romb**. The first two functions use Newton-Coates formulas of order 1 and 2 respectively to perform integration. These two functions can handle, non-equally-spaced samples. The trapezoidal rule approximates the function as a straight line between adjacent points, while Simpson's rule approximates the function between three adjacent points as a parabola.

If the samples are equally-spaced and the number of samples available is $2^k + 1$ for some integer $k$, then Romberg integration can be used to obtain high-precision estimates of the integral using the available samples. Romberg integration uses the trapezoid rule at step-sizes related by a power of two and then performs Richardson extrapolation on these estimates to approximate the integral with a higher-degree of accuracy. (A different interface to Romberg integration useful when the function can be provided is also available as **integrate.romberg**).

**6.4.4. Ordinary differential equations (integrate.odeint).** Integrating a set of ordinary differential equations (ODEs) given initial conditions is another useful example. The function **odeint** is available in SciPy for integrating a first-order vector differential equation:

$$\frac{d\mathbf{y}}{dt} = \mathbf{f}\left(\mathbf{y}, t\right),$$

given initial conditions $\mathbf{y}\left(0\right) = y_0$, where $\mathbf{y}$ is a length $N$ vector and $\mathbf{f}$ is a mapping from $\mathcal{R}^N$ to $\mathcal{R}^N$. A higher-order ordinary differential equation can always be reduced to a differential equation of this type by introducing intermediate derivatives into the $\mathbf{y}$ vector.

For example suppose it is desired to find the solution to the following second-order differential equation:

$$\frac{d^2w}{dz^2} - zw(z) = 0$$

with initial conditions $w\left(0\right) = \frac{1}{\sqrt[3]{3^2}\Gamma\left(\frac{2}{3}\right)}$ and $\frac{dw}{dz}\Big|_{z=0} = -\frac{1}{\sqrt[3]{3}\Gamma\left(\frac{1}{3}\right)}$. It is known that the solution to this differential equation with these boundary conditions is the Airy function

$$w = \mathrm{Ai}\left(z\right),$$

which gives a means to check the integrator using **special.airy.**

First, convert this ODE into standard form by setting $\mathbf{y} = \left[\frac{dw}{dz}, w\right]$ and $t = z$. Thus, the differential equation becomes

$$\frac{d\mathbf{y}}{dt} = \begin{bmatrix} ty_1 \\ y_0 \end{bmatrix} = \begin{bmatrix} 0 & t \\ 1 & 0 \end{bmatrix} \begin{bmatrix} y_0 \\ y_1 \end{bmatrix} = \begin{bmatrix} 0 & t \\ 1 & 0 \end{bmatrix} \mathbf{y}.$$

In other words,

$$\mathbf{f}\left(\mathbf{y}, t\right) = \mathbf{A}\left(t\right)\mathbf{y}.$$

As an interesting reminder, if $\mathbf{A}\left(t\right)$ commutes with $\int_0^t \mathbf{A}\left(\tau\right)\, d\tau$ under matrix multiplication, then this linear differential equation has an exact solution using the matrix exponential:

$$\mathbf{y}\left(t\right) = \exp\left(\int_0^t \mathbf{A}\left(\tau\right) d\tau\right)\mathbf{y}\left(0\right),$$

However, in this case, $\mathbf{A}\left(t\right)$ and its integral do not commute.

There are many optional inputs and outputs available when using odeint which can help tune the solver. These additional inputs and outputs are not needed much of the time, however, and the three required input arguments and the output solution suffice. The required inputs are the function defining the derivative, *fprime,* the initial conditions vector, *y0,* and the time points to obtain a solution, *t,* (with the initial value point as the

first element of this sequence). The output to **odeint** is a matrix where each row contains the solution vector at each requested time point (thus, the initial conditions are given in the first output row).

The following example illustrates the use of odeint including the usage of the **Dfun** option which allows the user to specify a gradient (with respect to $\mathbf{y}$) of the function, $\mathbf{f}(\mathbf{y}, t)$.

```
>>> from integrate import odeint
>>> from special import gamma, airy
>>> y1_0 = 1.0/3**(2.0/3.0)/gamma(2.0/3.0)
>>> y0_0 = -1.0/3**(1.0/3.0)/gamma(1.0/3.0)
>>> y0 = [y0_0, y1_0]
>>> def func(y, t):
        return [t*y[1],y[0]]

>>> def gradient(y,t):
        return [[0,t],[1,0]]

>>> x = arange(0,4.0, 0.01)
>>> t = x
>>> ychk = airy(x)[0]
>>> y = odeint(func, y0, t)
>>> y2 = odeint(func, y0, t, Dfun=gradient)

>>> import sys
>>> sys.float_output_precision = 6
>>> print ychk[:36:6]
[ 0.355028  0.339511  0.324068  0.308763  0.293658  0.278806]

>>> print y[:36:6,1]
[ 0.355028  0.339511  0.324067  0.308763  0.293658  0.278806]

>>> print y2[:36:6,1]
[ 0.355028  0.339511  0.324067  0.308763  0.293658  0.278806]
```

## 6.5. Optimization (optimize)

There are several classical optimization algorithms provided by SciPy in the **optimize** package. An overview of the module is available using **help** (or pydoc.help):

```
>>> info(optimize)
 Optimization Tools

A collection of general-purpose optimization routines.

  fmin        --  Nelder-Mead Simplex algorithm
                  (uses only function calls)
  fmin_powell --  Powell's (modified) level set method (uses only
                  function calls)
  fmin_bfgs   --  Quasi-Newton method (can use function and gradient)
  fmin_ncg    --  Line-search Newton Conjugate Gradient (can use
                  function, gradient and hessian).
  leastsq     --  Minimize the sum of squares of M equations in
                  N unknowns given a starting estimate.

 Scalar function minimizers

  fminbound   --  Bounded minimization of a scalar function.
  brent       --  1-D function minimization using Brent method.
  golden      --  1-D function minimization using Golden Section method
  bracket     --  Bracket a minimum (given two starting points)

Also a collection of general_purpose root-finding routines.
```

```
  fsolve      --  Non-linear multi-variable equation solver.

 Scalar function solvers

  brentq      --  quadratic interpolation Brent method
  brenth      --  Brent method (modified by Harris with
                    hyperbolic extrapolation)
  ridder      --  Ridder's method
  bisect      --  Bisection method
  newton      --  Secant method or Newton's method

  fixed_point -- Single-variable fixed-point solver.
```

The first four algorithms are unconstrained minimization algorithms (fmin: Nelder-Mead simplex, fmin_bfgs: BFGS, fmin_ncg: Newton Conjugate Gradient, and leastsq: Levenburg-Marquardt). The fourth algorithm only works for functions of a single variable but allows minimization over a specified interval. The last algorithm actually finds the roots of a general function of possibly many variables. It is included in the optimization package because at the (non-boundary) extreme points of a function, the gradient is equal to zero.

**6.5.1. Nelder-Mead Simplex algorithm (optimize.fmin).** The simplex algorithm is probably the simplest way to minimize a fairly well-behaved function. The simplex algorithm requires only function evaluations and is a good choice for simple minimization problems. However, because it does not use any gradient evaluations, it may take longer to find the minimum. To demonstrate the minimization function consider the problem of minimizing the Rosenbrock function of $N$ variables:

$$f(\mathbf{x}) = \sum_{i=1}^{N-1} 100\left(x_i - x_{i-1}^2\right)^2 + \left(1 - x_{i-1}\right)^2.$$

The minimum value of this function is 0 which is achieved when $x_i = 1$. This minimum can be found using the **fmin** routine as shown in the example below:

```
>>> from scipy.optimize import fmin
>>> def rosen(x):  # The Rosenbrock function
        return sum(100.0*(x[1:]-x[:-1]**2.0)**2.0 + (1-x[:-1])**2.0)

>>> x0 = [1.3, 0.7, 0.8, 1.9, 1.2]
>>> xopt = fmin(rosen, x0)
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 516
         Function evaluations: 825

>>> print xopt
[ 1.  1.  1.  1.  1.]
```

Another optimization algorithm that needs only function calls to find the minimum is Powell's method available as **optimize.fmin_powell**.

**6.5.2. Broyden-Fletcher-Goldfarb-Shanno algorithm (optimize.fmin_bfgs).** In order to converge more quickly to the solution, this routine uses the gradient of the objective function. If the gradient is not given by the user, then it is estimated using first-differences. The Broyden-Fletcher-Goldfarb-Shanno (BFGS) method typically requires fewer function calls than the simplex algorithm even when the gradient must be estimated.

To demonstrate this algorithm, the Rosenbrock function is again used. The gradient of the Rosenbrock function is the vector:

$$\begin{aligned}\frac{\partial f}{\partial x_j} &= \sum_{i=1}^{N} 200\left(x_i - x_{i-1}^2\right)\left(\delta_{i,j} - 2x_{i-1}\delta_{i-1,j}\right) - 2\left(1 - x_{i-1}\right)\delta_{i-1,j}. \\ &= 200\left(x_j - x_{j-1}^2\right) - 400x_j\left(x_{j+1} - x_j^2\right) - 2\left(1 - x_j\right).\end{aligned}$$

This expression is valid for the interior derivatives. Special cases are

$$\begin{aligned}\frac{\partial f}{\partial x_0} &= -400x_0\left(x_1 - x_0^2\right) - 2\left(1 - x_0\right), \\ \frac{\partial f}{\partial x_{N-1}} &= 200\left(x_{N-1} - x_{N-2}^2\right).\end{aligned}$$

A Python function which computes this gradient is constructed by the code-segment:

```
>>> def rosen_der(x):
        xm = x[1:-1]
        xm_m1 = x[:-2]
        xm_p1 = x[2:]
        der = zeros(x.shape,x.typecode())
        der[1:-1] = 200*(xm-xm_m1**2) - 400*(xm_p1 - xm**2)*xm - 2*(1-xm)
        der[0] = -400*x[0]*(x[1]-x[0]**2) - 2*(1-x[0])
        der[-1] = 200*(x[-1]-x[-2]**2)
        return der
```

The calling signature for the BFGS minimization algorithm is similar to **fmin** with the addition of the *fprime* argument. An example usage of **fmin_bfgs** is shown in the following example which minimizes the Rosenbrock function.

```
>>> from scipy.optimize import fmin_bfgs

>>> x0 = [1.3, 0.7, 0.8, 1.9, 1.2]
>>> xopt = fmin_bfgs(rosen, x0, fprime=rosen_der)
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 109
         Function evaluations: 262
         Gradient evaluations: 110
>>> print xopt
[ 1.  1.  1.  1.  1.]
```

**6.5.3. Newton-Conjugate-Gradient (optimize.fmin_ncg).** The method which requires the fewest function calls and is therefore often the fastest method to minimize functions of many variables is **fmin_ncg.** This method is a modified Newton's method and uses a conjugate gradient algorithm to (approximately) invert the local Hessian. Newton's method is based on fitting the function locally to a quadratic form:

$$f\left(\mathbf{x}\right) \approx f\left(\mathbf{x}_0\right) + \nabla f\left(\mathbf{x}_0\right) \cdot \left(\mathbf{x} - \mathbf{x}_0\right) + \frac{1}{2}\left(\mathbf{x} - \mathbf{x}_0\right)^T \mathbf{H}\left(\mathbf{x}_0\right)\left(\mathbf{x} - \mathbf{x}_0\right).$$

where $\mathbf{H}\left(\mathbf{x}_0\right)$ is a matrix of second-derivatives (the Hessian). If the Hessian is positive definite then the local minimum of this function can be found by setting the gradient of the quadratic form to zero, resulting in

$$\mathbf{x}_{\text{opt}} = \mathbf{x}_0 - \mathbf{H}^{-1}\nabla f.$$

The inverse of the Hessian is evaluted using the conjugate-gradient method. An example of employing this method to minimizing the Rosenbrock function is given below. To take full advantage of the NewtonCG method, a function which computes the Hessian must be provided. The Hessian matrix itself does not need to be constructed, only a vector which is the product of the Hessian with an arbitrary vector needs to be available to the minimization routine. As a result, the user can provide either a function to compute the Hessian matrix, or a function to compute the product of the Hessian with an arbitrary vector.

6.5.3.1. *Full Hessian example:* The Hessian of the Rosenbrock function is

$$\begin{aligned}H_{ij} = \frac{\partial^2 f}{\partial x_i \partial x_j} &= 200\left(\delta_{i,j} - 2x_{i-1}\delta_{i-1,j}\right) - 400x_i\left(\delta_{i+1,j} - 2x_i\delta_{i,j}\right) - 400\delta_{i,j}\left(x_{i+1} - x_i^2\right) + 2\delta_{i,j}, \\ &= \left(202 + 1200x_i^2 - 400x_{i+1}\right)\delta_{i,j} - 400x_i\delta_{i+1,j} - 400x_{i-1}\delta_{i-1,j},\end{aligned}$$

if $i, j \in [1, N-2]$ with $i, j \in [0, N-1]$ defining the $N \times N$ matrix. Other non-zero entries of the matrix are

$$\begin{aligned}
\frac{\partial^2 f}{\partial x_0^2} &= 1200x_0^2 - 400x_1 + 2, \\
\frac{\partial^2 f}{\partial x_0 \partial x_1} = \frac{\partial^2 f}{\partial x_1 \partial x_0} &= -400x_0, \\
\frac{\partial^2 f}{\partial x_{N-1} \partial x_{N-2}} = \frac{\partial^2 f}{\partial x_{N-2} \partial x_{N-1}} &= -400x_{N-2}, \\
\frac{\partial^2 f}{\partial x_{N-1}^2} &= 200.
\end{aligned}$$

For example, the Hessian when $N = 5$ is

$$\mathbf{H} = \begin{bmatrix}
1200x_0^2 - 400x_1 + 2 & -400x_0 & 0 & 0 & 0 \\
-400x_0 & 202 + 1200x_1^2 - 400x_2 & -400x_1 & 0 & 0 \\
0 & -400x_1 & 202 + 1200x_2^2 - 400x_3 & -400x_2 & 0 \\
0 & & -400x_2 & 202 + 1200x_3^2 - 400x_4 & -400x_3 \\
0 & 0 & 0 & -400x_3 & 200
\end{bmatrix}.$$

The code which computes this Hessian along with the code to minimize the function using **fmin_ncg** is shown in the following example:

```
>>> from scipy.optimize import fmin_ncg
>>> def rosen_hess(x):
        x = asarray(x)
        H = diag(-400*x[:-1],1) - diag(400*x[:-1],-1)
        diagonal = zeros(len(x),x.typecode())
        diagonal[0] = 1200*x[0]-400*x[1]+2
        diagonal[-1] = 200
        diagonal[1:-1] = 202 + 1200*x[1:-1]**2 - 400*x[2:]
        H = H + diag(diagonal)
        return H

>>> x0 = [1.3, 0.7, 0.8, 1.9, 1.2]
>>> xopt = fmin_ncg(rosen, x0, rosen_der, fhess=rosen_hess)
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 19
         Function evaluations: 40
         Gradient evaluations: 19
         Hessian evaluations: 19
>>> print xopt
[ 0.9999  0.9999  0.9998  0.9996  0.9991]
```

6.5.3.2. *Hessian product example:* For larger minimization problems, storing the entire Hessian matrix can consume considerable time and memory. The Newton-CG algorithm only needs the product of the Hessian times an arbitrary vector. As a result, the user can supply code to compute this product rather than the full Hessian by setting the *fhess_p* keyword to the desired function. The fhess_p function should take the minimization vector as the first argument and the arbitrary vector as the second argument. Any extra arguments passed to the function to be minimized will also be passed to this function. If possible, using Newton-CG with the hessian product option is probably the fastest way to minimize the function.

In this case, the product of the Rosenbrock Hessian with an arbitrary vector is not difficult to compute. If $\mathbf{p}$ is the arbitrary vector, then $\mathbf{H}(\mathbf{x})\mathbf{p}$ has elements:

$$\mathbf{H}(\mathbf{x})\mathbf{p} = \begin{bmatrix}
\left(1200x_0^2 - 400x_1 + 2\right)p_0 - 400x_0p_1 \\
\vdots \\
-400x_{i-1}p_{i-1} + \left(202 + 1200x_i^2 - 400x_{i+1}\right)p_i - 400x_ip_{i+1} \\
\vdots \\
-400x_{N-2}p_{N-2} + 200p_{N-1}
\end{bmatrix}.$$

Code which makes use of the *fhess_p* keyword to minimize the Rosenbrock function using **fmin_ncg** follows:

```
>>> from scipy.optimize import fmin_ncg
>>> def rosen_hess_p(x,p):
        x = asarray(x)
        Hp = zeros(len(x),x.typecode())
        Hp[0] = (1200*x[0]**2 - 400*x[1] + 2)*p[0] - 400*x[0]*p[1]
        Hp[1:-1] = -400*x[:-2]*p[:-2]+(202+1200*x[1:-1]**2-400*x[2:])*p[1:-1] \
                   -400*x[1:-1]*p[2:]
        Hp[-1] = -400*x[-2]*p[-2] + 200*p[-1]
        return Hp

>>> x0 = [1.3, 0.7, 0.8, 1.9, 1.2]
>>> xopt = fmin_ncg(rosen, x0, rosen_der, fhess_p=rosen_hess_p)
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 20
         Function evaluations: 42
         Gradient evaluations: 20
         Hessian evaluations: 44
>>> print xopt
[ 1.      1.      1.      0.9999  0.9999]
```

**6.5.4. Least-square fitting (minimize.leastsq).** All of the previously-explained minimization procedures can be used to solve a least-squares problem provided the appropriate objective function is constructed. For example, suppose it is desired to fit a set of data $\{\mathbf{x}_i, \mathbf{y}_i\}$ to a known model, $\mathbf{y} = \mathbf{f}(\mathbf{x}, \mathbf{p})$ where $\mathbf{p}$ is a vector of parameters for the model that need to be found. A common method for determining which parameter vector gives the best fit to the data is to minimize the sum of squares of the residuals. The residual is usually defined for each observed data-point as

$$e_i(\mathbf{p}, \mathbf{y}_i, \mathbf{x}_i) = \|\mathbf{y}_i - \mathbf{f}(\mathbf{x}_i, \mathbf{p})\|.$$

An objective function to pass to any of the previous minization algorithms to obtain a least-squares fit is.

$$J(\mathbf{p}) = \sum_{i=0}^{N-1} e_i^2(\mathbf{p}).$$

The **leastsq** algorithm performs this squaring and summing of the residuals automatically. It takes as an input argument the vector function $\mathbf{e}(\mathbf{p})$ and returns the value of $\mathbf{p}$ which minimizes $J(\mathbf{p}) = \mathbf{e}^T\mathbf{e}$ directly. The user is also encouraged to provide the Jacobian matrix of the function (with derivatives down the columns or across the rows). If the Jacobian is not provided, it is estimated.

An example should clarify the usage. Suppose it is believed some measured data follow a sinusoidal pattern

$$y_i = A\sin(2\pi k x_i + \theta)$$

where the parameters $A$, $k$, and $\theta$ are unknown. The residual vector is

$$e_i = |y_i - A\sin(2\pi k x_i + \theta)|.$$

By defining a function to compute the residuals and (selecting an appropriate starting position), the least-squares fit routine can be used to find the best-fit parameters $\hat{A}$, $\hat{k}$, $\hat{\theta}$. This is shown in the following example and a plot of the results is shown in Figure 6.5.1.

```
>>> x = arange(0,6e-2,6e-2/30)
>>> A,k,theta = 10, 1.0/3e-2, pi/6
>>> y_true = A*sin(2*pi*k*x+theta)
>>> y_meas = y_true + 2*randn(len(x))

>>> def residuals(p, y, x):
        A,k,theta = p
        err = y-A*sin(2*pi*k*x+theta)
        return err

>>> def peval(x, p):
        return p[0]*sin(2*pi*p[1]*x+p[2])

>>> p0 = [8, 1/2.3e-2, pi/3]
```

```
>>> print array(p0)
[  8.      43.4783   1.0472]

>>> from scipy.optimize import leastsq
>>> plsq = leastsq(residuals, p0, args=(y_meas, x))
>>> print plsq[0]
[ 10.9437  33.3605   0.5834]

>>> print array([A, k, theta])
[ 10.      33.3333   0.5236]

>>> from xplt import *    # Only on X-windows systems
>>> plot(x,peval(x,plsq[0]),x,y_meas,'o',x,y_true)
>>> title('Least-squares fit to noisy data')
>>> legend(['Fit', 'Noisy', 'True'])
>>> gist.eps('leastsqfit')   # Make epsi file.
```

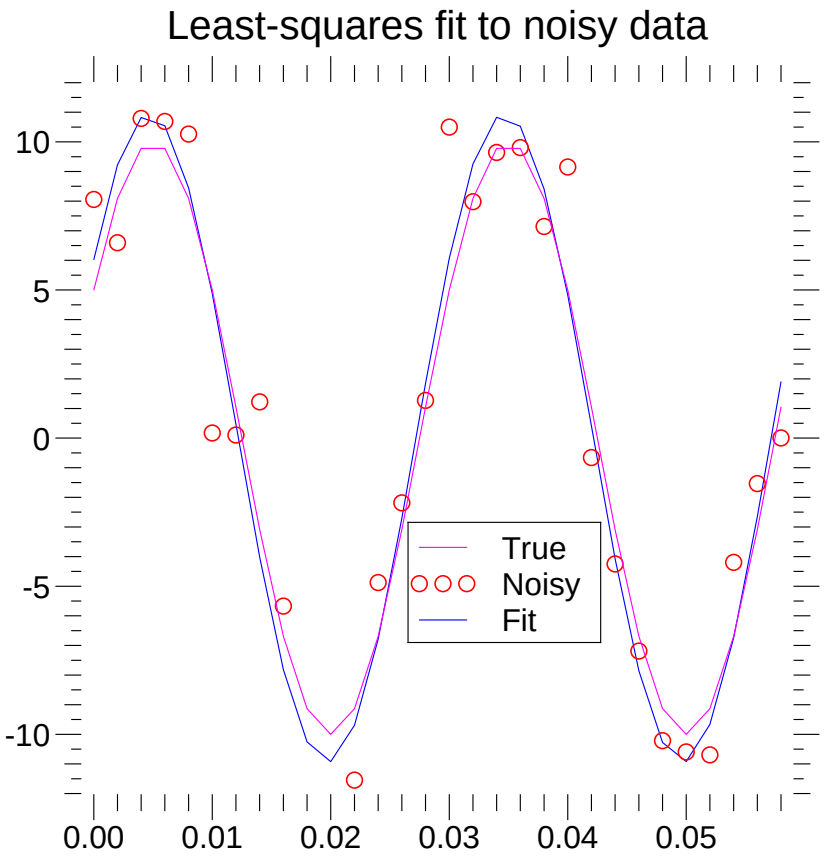



FIGURE 6.5.1. Least-square fitting to noisy data using **scipy.optimize.leastsq**

**6.5.5. Scalar function minimizers.** Often only the minimum of a scalar function is needed (a scalar function is one that takes a scalar as input and returns a scalar output). In these circumstances, other optimization techniques have been developed that can work faster.

6.5.5.1. *Unconstrained minimization (optimize.brent).* There are actually two methods that can be used to minimize a scalar function (**brent** and **golden**), but **golden** is included only for academic purposes and should rarely be used. The brent method uses Brent's algorithm for locating a minimum. Optimally a bracket should be given which contains the minimum desired. A bracket is a triple $(a, b, c)$ such that $f(a) > f(b) < f(c)$ and $a < b < c$. If this is not given, then alternatively two starting points can be chosen and a bracket will be found from these points using a simple marching algorithm. If these two starting points are not provided 0 and 1

will be used (this may not be the right choice for your function and result in an unexpected minimum being returned).

6.5.5.2. *Bounded minimization (optimize.fminbound).* Thus far all of the minimization routines described have been unconstrained minimization routines. Very often, however, there are constraints that can be placed on the solution space before minimization occurs. The **fminbound** function is an example of a constrained minimization procedure that provides a rudimentary interval constraint for scalar functions. The interval constraint allows the minimization to occur only between two fixed endpoints.

For example, to find the minimum of $J_1(x)$ near $x = 5$, **fminbound** can be called using the interval $[4, 7]$ as a constraint. The result is $x_{\min} = 5.3314$:

```
>>> from scipy.special import j1

>>> from scipy.optimize import fminbound
>>> xmin = fminbound(j1, 4, 7)
>>> print xmin
5.33144184241
```

### 6.5.6. Root finding.

6.5.6.1. *Sets of equations.* To find the roots of a polynomial, the command **roots** is useful. To find a root of a set of non-linear equations, the command **optimize.fsolve** is needed. For example, the following example finds the roots of the single-variable transcendental equation

$$x + 2\cos(x) = 0,$$

and the set of non-linear equations

$$\begin{aligned} x_0 \cos(x_1) &= 4, \\ x_0 x_1 - x_1 &= 5. \end{aligned}$$

The results are $x = -1.0299$ and $x_0 = 6.5041$, $x_1 = 0.9084$.

```
>>> def func(x):
        return x + 2*cos(x)

>>> def func2(x):
        out = [x[0]*cos(x[1]) - 4]
        out.append(x[1]*x[0] - x[1] - 5)
        return out

>>> from optimize import fsolve
>>> x0 = fsolve(func, 0.3)
>>> print x0
-1.02986652932

>>> x02 = fsolve(func2, [1, 1])
>>> print x02
[ 6.5041  0.9084]
```

6.5.6.2. *Scalar function root finding.* If one has a single-variable equation, there are four different root finder algorithms that can be tried. Each of these root finding algorithms requires the endpoints of an interval where a root is suspected (because the function changes signs). In general **brentq** is the best choice, but the other methods may be useful in certain circumstances or for academic purposes.

6.5.6.3. *Fixed-point solving.* A problem closely related to finding the zeros of a function is the problem of finding a fixed-point of a function. A fixed point of a function is the point at which evaluation of the function returns the point: $g(x) = x$. Clearly the fixed point of $g$ is the root of $f(x) = g(x) - x$. Equivalently, the root of $f$ is the fixed_point of $g(x) = f(x) + x$. The routine **fixed_point** provides a simple iterative method using Aitkens sequence acceleration to estimate the fixed point of $g$ given a starting point.

## 6.6. Interpolation (interpolate)

There are two general interpolation facilities available in SciPy. The first facility is an interpolation class which performs linear 1-dimensional interpolation. The second facility is based on the FORTRAN library FITPACK and provides functions for 1- and 2-dimensional (smoothed) cubic-spline interpolation.

**6.6.1. Linear 1-d interpolation (interpolate.linear_1d).** The linear_1d class in scipy.interpolate is a convenient method to create a function based on fixed data points which can be evaluated anywhere within the domain defined by the given data using linear interpolation. An instance of this class is created by passing the 1-d vectors comprising the data. The instance of this class defines a *__call__* method and can therefore by treated like a function which interpolates between known data values to obtain unknown values (it even has a docstring for help). Behavior at the boundary can be specified at instantiation time. The following example demonstrates it's use.

```
>>> x = arange(0,10)
>>> y = exp(-x/3.0)
>>> f = interpolate.linear_1d(x,y)
>>> help(f)
Instance of class:  linear_1d

 <name>(x_new)

Find linearly interpolated y_new = <name>(x_new).

Inputs:
  x_new -- New independent variables.

Outputs:
  y_new -- Linearly interpolated values corresponding to x_new.

>>> xnew = arange(0,9,0.1)
>>> xplt.plot(x,y,'x',xnew,f(xnew),'.')
```

Figure shows the result:

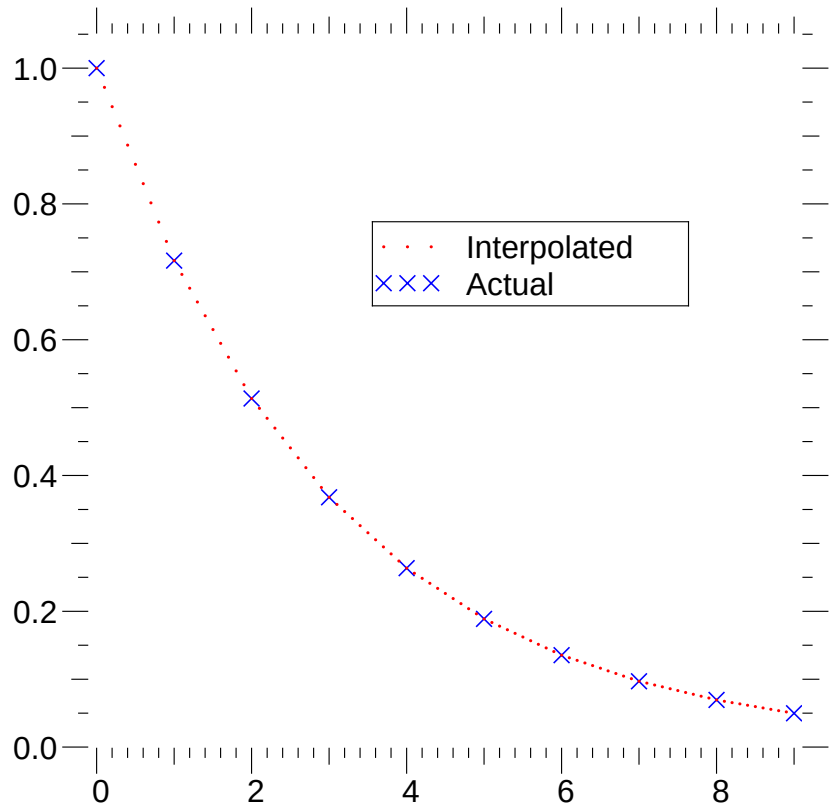



FIGURE 6.6.1. One-dimensional interpolation using the class **interpolate.linear_1d.**

**6.6.2. Spline interpolation in 1-d (interpolate.splXXX).** Spline interpolation requires two essential steps: (1) a spline representation of the curve is computed, and (2) the spline is evaluated at the desired points. In order to find the spline representation, there are two different was to represent a curve and obtain (smoothing) spline coefficients: directly and parametrically. The direct method finds the spline representation of a curve in a two-dimensional plane using the function **interpolate.splrep.** The first two arguments are the only ones required, and these provide the $x$ and $y$ components of the curve. The normal output is a 3-tuple, $(t, c, k)$, containing the knot-points, $t$, the coefficients $c$ and the order $k$ of the spline. The default spline order is cubic, but this can be changed with the input keyword, $k$.

For curves in $N$-dimensional space the function **interpolate.splprep** allows defining the curve parametrically. For this function only 1 input argument is required. This input is a list of $N$-arrays representing the curve in $N$-dimensional space. The length of each array is the number of curve points, and each array provides one component of the $N$-dimensional data point. The parameter variable is given with the keword argument, $u$, which defaults to an equally-spaced monotonic sequence between 0 and 1. The default output consists of two objects: a 3-tuple, $(t, c, k)$, containing the spline representation and the parameter variable $u$.

The keyword argument, $s$, is used to specify the amount of smoothing to perform during the spline fit. The default value of $s$ is $s = m - \sqrt{2m}$ where $m$ is the number of data-points being fit. Therefore, **if no smoothing is desired a value of s = 0 should be passed to the routines.**

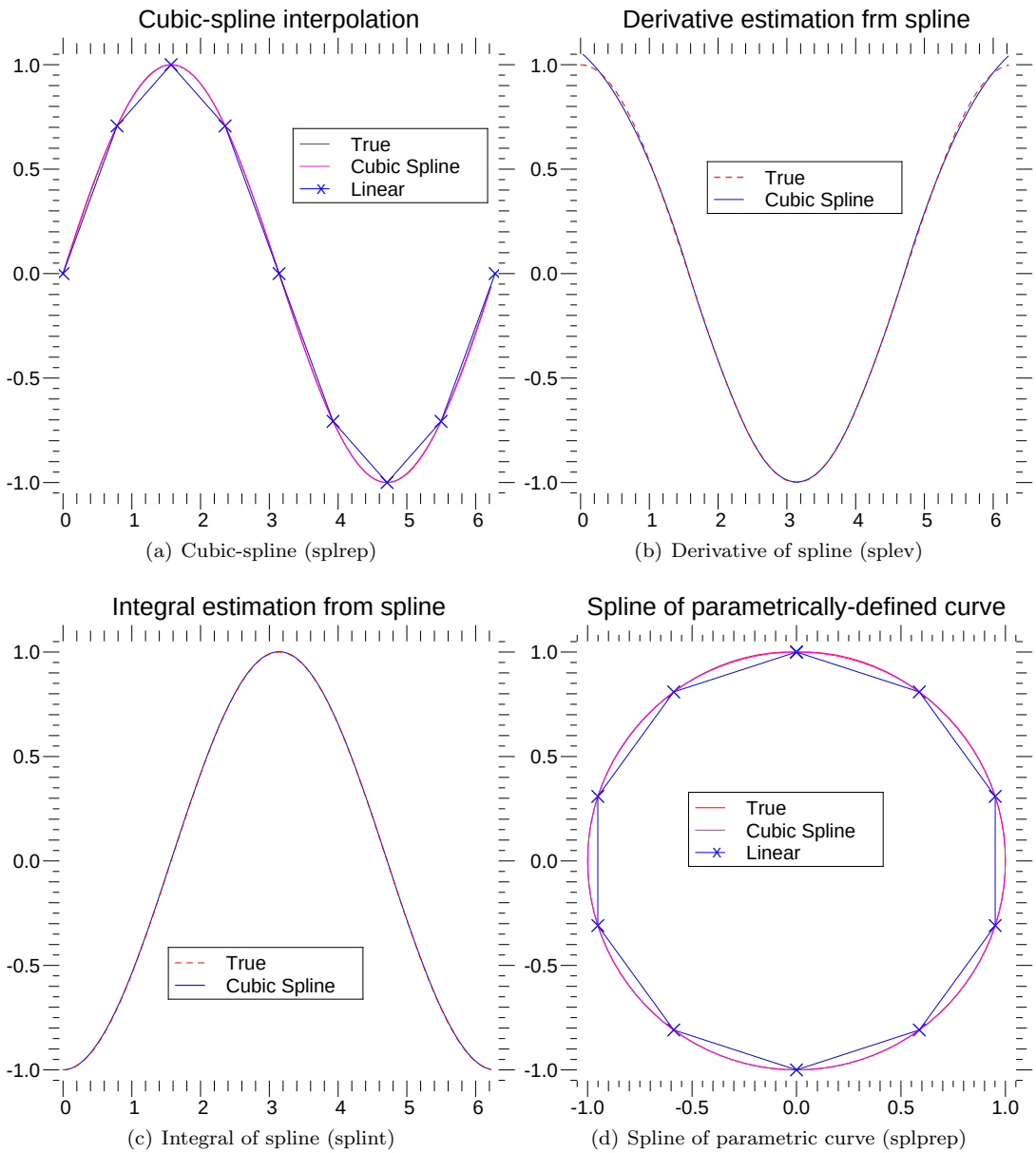
Once the spline representation of the data has been determined, functions are available for evaluating the spline (**interpolate.splev)** and its derivatives (**interpolate.splev, interpolate.splade**) at any point and the integral of the spline between any two points (**interpolate.splint)**. In addition, for cubic splines ($k = 3$) with 8 or more knots, the roots of the spline can be estimated (**interpolate.sproot)**. These functions are demonstrated in the example that follows (see also Figure 6.6.2).

```
>>> # Cubic-spline
>>> x = arange(0,2*pi+pi/4,2*pi/8)
>>> y = sin(x)
>>> tck = interpolate.splrep(x,y,s=0)
>>> xnew = arange(0,2*pi,pi/50)
>>> ynew = interpolate.splev(xnew,tck,der=0)
>>> xplt.plot(x,y,'x',xnew,ynew,xnew,sin(xnew),x,y,'b')
>>> xplt.legend(['Linear','Cubic Spline', 'True'],['b-x','m','r'])
>>> xplt.limits(-0.05,6.33,-1.05,1.05)
>>> xplt.title('Cubic-spline interpolation')
>>> xplt.eps('interp_cubic')

>>> # Derivative of spline
>>> yder = interpolate.splev(xnew,tck,der=1)
>>> xplt.plot(xnew,yder,xnew,cos(xnew),'|')
>>> xplt.legend(['Cubic Spline', 'True'])
>>> xplt.limits(-0.05,6.33,-1.05,1.05)
>>> xplt.title('Derivative estimation from spline')
>>> xplt.eps('interp_cubic_der')

>>> # Integral of spline
>>> def integ(x,tck,constant=-1):
>>>     x = asarray_1d(x)
>>>     out = zeros(x.shape, x.typecode())
>>>     for n in xrange(len(out)):
>>>         out[n] = interpolate.splint(0,x[n],tck)
>>>     out += constant
>>>     return out
>>>
>>> yint = integ(xnew,tck)
>>> xplt.plot(xnew,yint,xnew,-cos(xnew),'|')
>>> xplt.legend(['Cubic Spline', 'True'])
>>> xplt.limits(-0.05,6.33,-1.05,1.05)
>>> xplt.title('Integral estimation from spline')
>>> xplt.eps('interp_cubic_int')

>>> # Roots of spline
```




(a) Cubic-spline (splrep) (b) Derivative of spline (splev)

(c) Integral of spline (splint) (d) Spline of parametric curve (splprep)

FIGURE 6.6.2. Examples of using cubic-spline interpolation.

```
>>> print interpolate.sproot(tck)
[ 0.      3.1416]

>>> # Parametric spline
>>> t = arange(0,1.1,.1)
>>> x = sin(2*pi*t)
>>> y = cos(2*pi*t)
>>> tck,u = interpolate.splprep([x,y],s=0)
>>> unew = arange(0,1.01,0.01)
>>> out = interpolate.splev(unew,tck)
>>> xplt.plot(x,y,'x',out[0],out[1],sin(2*pi*unew),cos(2*pi*unew),x,y,'b')
>>> xplt.legend(['Linear','Cubic Spline', 'True'],['b-x','m','r'])
>>> xplt.limits(-1.05,1.05,-1.05,1.05)
```

```
>>> xplt.title('Spline of parametrically-defined curve')
>>> xplt.eps('interp_cubic_param')
```

**6.6.3. Two-dimensionsal spline representation (interpolate.bisplrep).** For (smooth) spline-fitting to a two dimensional surface, the function **interpolate.bisplrep** is available. This function takes as required inputs the **1-D** arrays *x, y,* and $z$ which represent points on the surface $z = f(x, y)$. The default output is a list $[tx, ty, c, kx, ky]$ whose entries represent respectively, the components of the knot positions, the coefficients of the spline, and the order of the spline in each coordinate. It is convenient to hold this list in a single object, *tck,* so that it can be passed easily to the function **interpolate.bisplev.** The keyword, $s$, can be used to change the amount of smoothing performed on the data while determining the appropriate spline. The default value is $s = m - \sqrt{2m}$ where $m$ is the number of data points in the *x, y,* and $z$ vectors. As a result, if no smoothing is desired, then $s = 0$ should be passed to **interpolate.bisplrep.**

To evaluate the two-dimensional spline and it's partial derivatives (up to the order of the spline), the function **interpolate.bisplev** is required. This function takes as the first two arguments **two 1-D arrays** whose cross-product specifies the domain over which to evaluate the spline. The third argument is the *tck* list returned from **interpolate.bisplrep.** If desired, the fourth and fifth arguments provide the orders of the partial derivative in the $x$ and $y$ direction respectively.

It is important to note that two dimensional interpolation should not be used to find the spline representation of images. The algorithm used is not amenable to large numbers of input points. The signal processing toolbox contains more appropriate algorithms for finding the spline representation of an image. The two dimensional interpolation commands are intended for use when interpolating a two dimensional function as shown in the example that follows (See also Figure 6.6.3). This example uses the **mgrid** command in SciPy which is useful for defining a "mesh-grid" in many dimensions. (See also the **ogrid** command if the full-mesh is not needed). The number of output arguments and the number of dimensions of each argument is determined by the number of indexing objects passed in **mgrid[].**

```
>>> # Define function over sparse 20x20 grid
>>> x,y = mgrid[-1:1:20j,-1:1:20j]
>>> z = (x+y)*exp(-6.0*(x*x+y*y))
>>> xplt.surf(z,x,y,shade=1,palette='rainbow')
>>> xplt.title3("Sparsely sampled function.")
>>> xplt.eps("2d_func")

>>> # Interpolate function over new 70x70 grid
>>> xnew,ynew = mgrid[-1:1:70j,-1:1:70j]
>>> tck = interpolate.bisplrep(x,y,z,s=0)
>>> znew = interpolate.bisplev(xnew[:,0],ynew[0,:],tck)
>>> xplt.surf(znew,xnew,ynew,shade=1,palette='rainbow')
>>> xplt.title3("Interpolated function.")
>>> xplt.eps("2d_interp")
```

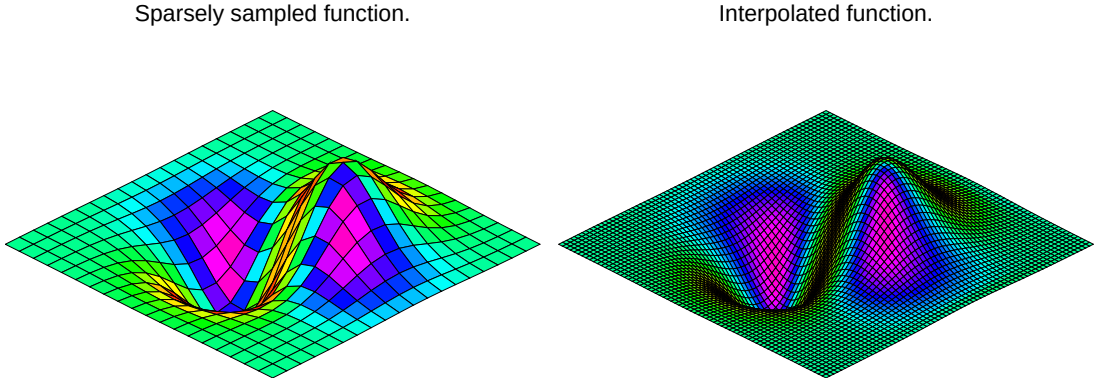



FIGURE 6.6.3. Example of two-dimensional spline interpolation.

## 6.7. Signal Processing (signal)

The signal processing toolbox currently contains some filtering functions, a limited set of filter design tools, and a few B-spline interpolation algorithms for one- and two-dimensional data. While the B-spline algorithms could technically be placed under the interpolation category, they are included here because they only work with equally-spaced data and make heavy use of filter-theory and transfer-function formalism to provide a fast B-spline transform. To understand this section you will need to understand that a signal in SciPy is an array of real or complex numbers.

**6.7.1. B-splines.** A B-spline is an approximation of a continuous function over a finite-domain in terms of B-spline coefficients and knot points. If the knot-points are equally spaced with spacing $\Delta x$, then the B-spline approximation to a 1-dimensional function is the finite-basis expansion.

$$y(x) \approx \sum_j c_j \beta^o \left( \frac{x}{\Delta x} - j \right).$$

In two dimensions with knot-spacing $\Delta x$ and $\Delta y$, the function representation is

$$z(x,y) \approx \sum_j \sum_k c_{jk} \beta^o \left( \frac{x}{\Delta x} - j \right) \beta^o \left( \frac{y}{\Delta y} - k \right).$$

In these expressions, $\beta^o(\cdot)$ is the space-limited B-spline basis function of order, $o$. The requirement of equally-spaced knot-points and equally-spaced data points, allows the development of fast (inverse-filtering) algorithms for determining the coefficients, $c_j$, from sample-values, $y_n$. Unlike the general spline interpolation algorithms, these algorithms can quickly find the spline coefficients for large images.

The advantage of representing a set of samples via B-spline basis functions is that continuous-domain operators (derivatives, re-sampling, integral, etc.) which assume that the data samples are drawn from an underlying continuous function can be computed with relative ease from the spline coefficients. For example, the second-derivative of a spline is

$$y''(x) = \frac{1}{\Delta x^2} \sum_j c_j \beta^{o\prime\prime} \left( \frac{x}{\Delta x} - j \right).$$

Using the property of B-splines that

$$\frac{d^2 \beta^o(w)}{dw^2} = \beta^{o-2}(w+1) - 2\beta^{o-2}(w) + \beta^{o-2}(w-1)$$

it can be seen that

$$y''(x) = \frac{1}{\Delta x^2} \sum_j c_j \left[ \beta^{o-2} \left( \frac{x}{\Delta x} - j + 1 \right) - 2\beta^{o-2} \left( \frac{x}{\Delta x} - j \right) + \beta^{o-2} \left( \frac{x}{\Delta x} - j - 1 \right) \right].$$

If $o = 3$, then at the sample points,

$$\begin{aligned} \Delta x^2 \, y'(x)\big|_{x=n\Delta x} &= \sum_j c_j \delta_{n-j+1} - 2c_j \delta_{n-j} + c_j \delta_{n-j-1}, \\ &= c_{n+1} - 2c_n + c_{n-1}. \end{aligned}$$

Thus, the second-derivative signal can be easily calculated from the spline fit. if desired, smoothing splines can be found to make the second-derivative less sensitive to random-errors.

The savvy reader will have already noticed that the data samples are related to the knot coefficients via a convolution operator, so that simple convolution with the sampled B-spline function recovers the original data from the spline coefficients. The output of convolutions can change depending on how boundaries are handled (this becomes increasingly more important as the number of dimensions in the data-set increases). The algorithms relating to B-splines in the signal-processing sub package assume mirror-symmetric boundary conditions. Thus, spline coefficients are computed based on that assumption, and data-samples can be recovered exactly from the spline coefficients by assuming them to be mirror-symmetric also.

Currently the package provides functions for determining second- and third-order cubic spline coefficients from equally spaced samples in one- and two-dimensions (**signal.qspline1d, signal.qspline2d, signal.cspline1d, signal.cspline2d**). The package also supplies a function (**signal.bspline**) for evaluating the bspline basis function, $\beta^o(x)$ for arbitrary order and $x$. For large $o$, the B-spline basis function can be approximated well by a zero-mean Gaussian function with standard-deviation equal to $\sigma_o = (o+1)/12$:

$$\beta^o(x) \approx \frac{1}{\sqrt{2\pi\sigma_o^2}} \exp\left( -\frac{x^2}{2\sigma_o} \right).$$

A function to compute this Gaussian for arbitrary $x$ and $o$ is also available (**signal.gauss_spline**). The following code and Figure uses spline-filtering to compute an edge-image (the second-derivative of a smoothed

spline) of Lena's face which is an array returned by the command **lena().** The command **signal.sepfir2d** was used to apply a separable two-dimensional FIR filter with mirror-symmetric boundary conditions to the spline coefficients. This function is ideally suited for reconstructing samples from spline coefficients and is faster than **signal.convolve2d** which convolves arbitrary two-dimensional filters and allows for choosing mirror-symmetric boundary conditions.

```
>>> image = lena().astype(Float32)
>>> derfilt = array([1.0,-2,1.0],Float32)
>>> ck = signal.cspline2d(image,8.0)
>>> deriv = signal.sepfir2d(ck, derfilt, [1]) + \
>>>         signal.sepfir2d(ck, [1], derfilt)
>>>
>>> ## Alternatively we could have done:
>>> ## laplacian = array([[0,1,0],[1,-4,1],[0,1,0]],Float32)
>>> ## deriv2 = signal.convolve2d(ck,laplacian,mode='same',boundary='symm')
>>>
>>> xplt.imagesc(image[::-1]) # flip image so it looks right-side up.
>>> xplt.title('Original image')
>>> xplt.eps('lena_image')
>>> xplt.imagesc(deriv[::-1])
>>> xplt.title('Output of spline edge filter')
>>> xplt.eps('lena_edge')
```

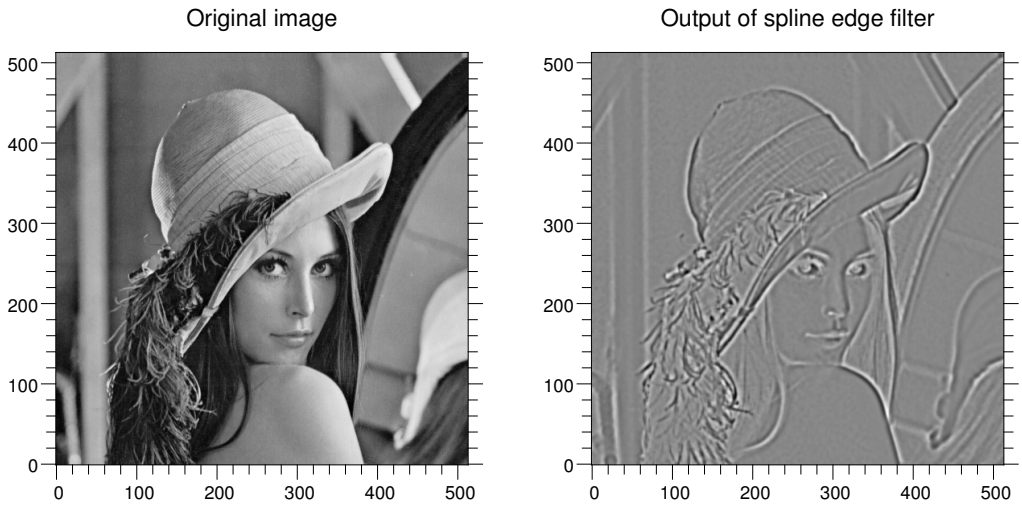



FIGURE 6.7.1. Example of using smoothing splines to filter images.

**6.7.2. Filtering.** Filtering is a generic name for any system that modifies an input signal in some way. In SciPy a signal can be thought of as a Numeric array. There are different kinds of filters for different kinds of operations. There are two broad kinds of filtering operations: linear and non-linear. Linear filters can always be reduced to multiplication of the flattened Numeric array by an appropriate matrix resulting in another flattened Numeric array. Of course, this is not usually the best way to compute the filter as the matrices and vectors involved may be huge. For example filtering a $512 \times 512$ image with this method would require multiplication of a $512^2 x 512^2$matrix with a $512^2$ vector. Just trying to store the $512^2 \times 512^2$ matrix using a standard Numeric array would require $68,719,476,736$ elements. At 4 bytes per element this would require 256GB of memory. In most applications most of the elements of this matrix are zero and a different method for computing the output of the filter is employed.

6.7.2.1. *Convolution/Correlation.* Many linear filters also have the property of shift-invariance. This means that the filtering operation is the same at different locations in the signal and it implies that the filtering matrix

can be constructed from knowledge of one row (or column) of the matrix alone. In this case, the matrix multiplication can be accomplished using Fourier transforms.

Let $x[n]$ define a one-dimensional signal indexed by the integer $n$. Full convolution of two one-dimensional signals can be expressed as

$$y[n] = \sum_{k=-\infty}^{\infty} x[k]\, h[n-k].$$

This equation can only be implemented directly if we limit the sequences to finite support sequences that can be stored in a computer, choose $n = 0$ to be the starting point of both sequences, let $K+1$ be that value for which $y[n] = 0$ for all $n > K+1$ and $M+1$ be that value for which $x[n] = 0$ for all $n > M+1$, then the discrete convolution expression is

$$y[n] = \sum_{k=\max(n-M,0)}^{\min(n,K)} x[k]\, h[n-k].$$

For convenience assume $K \geq M$. Then, more explicitly the output of this operation is

$$\begin{aligned}
y[0] &= x[0]\,h[0] \\
y[1] &= x[0]\,h[1] + x[1]\,h[0] \\
y[2] &= x[0]\,h[2] + x[1]\,h[1] + x[2]\,h[0] \\
&\vdots \quad \vdots \quad \vdots \\
y[M] &= x[0]\,h[M] + x[1]\,h[M-1] + \cdots + x[M]\,h[0] \\
y[M+1] &= x[1]\,h[M] + x[2]\,h[M-1] + \cdots + x[M+1]\,h[0] \\
&\vdots \quad \vdots \quad \vdots \\
y[K] &= x[K-M]\,h[M] + \cdots + x[K]\,h[0] \\
y[K+1] &= x[K+1-M]\,h[M] + \cdots + x[K]\,h[1] \\
&\vdots \quad \vdots \quad \vdots \\
y[K+M-1] &= x[K-1]\,h[M] + x[K]\,h[M-1] \\
y[K+M] &= x[K]\,h[M].
\end{aligned}$$

Thus, the full discrete convolution of two finite sequences of lengths $K+1$ and $M+1$ respectively results in a finite sequence of length $K+M+1 = (K+1)+(M+1)-1$.

One dimensional convolution is implemented in SciPy with the function `signal.convolve`. This function takes as inputs the signals $x$, $h$, and an optional flag and returns the signal $y$. The optional flag allows for specification of which part of the output signal to return. The default value of 'full' returns the entire signal. If the flag has a value of 'same' then only the middle $K$ values are returned starting at $y\left[\left\lfloor \frac{M-1}{2} \right\rfloor\right]$ so that the output has the same length as the largest input. If the flag has a value of 'valid' then only the middle $K-M+1 = (K+1)-(M+1)+1$ output values are returned where $z$ depends on all of the values of the smallest input from $h[0]$ to $h[M]$. In other words only the values $y[M]$ to $y[K]$ inclusive are returned.

This same function `signal.convolve` can actually take $N$-dimensional arrays as inputs and will return the $N$-dimensional convolution of the two arrays. The same input flags are available for that case as well.

Correlation is very similar to convolution except for the minus sign becomes a plus sign. Thus

$$w[n] = \sum_{k=-\infty}^{\infty} y[k]\, x[n+k]$$

is the (cross) correlation of the signals $y$ and $x$. For finite-length signals with $y[n] = 0$ outside of the range $[0, K]$ and $x[n] = 0$ outside of the range $[0, M]$, the summation can simplify to

$$w[n] = \sum_{k=\max(0,-n)}^{\min(K,M-n)} y[k]\, x[n+k].$$

Assuming again that $K \geq M$ this is

$$\begin{aligned}
w[-K] &= y[K]\,x[0] \\
w[-K+1] &= y[K-1]\,x[0] + y[K]\,x[1] \\
&\vdots \\
w[M-K] &= y[K-M]\,x[0] + y[K-M+1]\,x[1] + \cdots + y[K]\,x[M] \\
w[M-K+1] &= y[K-M-1]\,x[0] + \cdots + y[K-1]\,x[M] \\
&\vdots \\
w[-1] &= y[1]\,x[0] + y[2]\,x[1] + \cdots + y[M+1]\,x[M] \\
w[0] &= y[0]\,x[0] + y[1]\,x[1] + \cdots + y[M]\,x[M] \\
w[1] &= y[0]\,x[1] + y[1]\,x[2] + \cdots + y[M-1]\,x[M] \\
w[2] &= y[0]\,x[2] + y[1]\,x[3] + \cdots + y[M-2]\,x[M] \\
&\vdots \\
w[M-1] &= y[0]\,x[M-1] + y[1]\,x[M] \\
w[M] &= y[0]\,x[M].
\end{aligned}$$

The SciPy function `signal.correlate` implements this operation. Equivalent flags are available for this operation to return the full $K+M+1$ length sequence ('full') or a sequence with the same size as the largest sequence starting at $w\left[-K+\left\lfloor\frac{M-1}{2}\right\rfloor\right]$ ('same') or a sequence where the values depend on all the values of the smallest sequence ('valid'). This final option returns the $K-M+1$ values $w[M-K]$ to $w[0]$ inclusive.

The function **signal.correlate** can also take arbitrary $N$-dimensional arrays as input and return the $N$-dimensional convolution of the two arrays on output.

When $N=2$, **signal.correlate** and/or **signal.convolve** can be used to construct arbitrary image filters to perform actions such as blurring, enhancing, and edge-detection for an image.

Convolution is mainly used for filtering when one of the signals is much smaller than the other ($K \gg M$), otherwise linear filtering is more easily accomplished in the frequency domain (see Fourier Transforms).

6.7.2.2. *Difference-equation filtering.* A general class of linear one-dimensional filters (that includes convolution filters) are filters described by the difference equation

$$\sum_{k=0}^{N} a_k y[n-k] = \sum_{k=0}^{M} b_k x[n-k]$$

where $x[n]$ is the input sequence and $y[n]$ is the output sequence. If we assume initial rest so that $y[n]=0$ for $n<0$, then this kind of filter can be implemented using convolution. However, the convolution filter sequence $h[n]$ could be infinite if $a_k \neq 0$ for $k \geq 1$. In addition, this general class of linear filter allows initial conditions to be placed on $y[n]$ for $n<0$ resulting in a filter that cannot be expressed using convolution.

The difference equation filter can be thought of as finding $y[n]$ recursively in terms of it's previous values

$$a_0 y[n] = -a_1 y[n-1] - \cdots - a_N y[n-N] + \cdots + b_0 x[n] + \cdots + b_M x[n-M].$$

Often $a_0 = 1$ is chosen for normalization. The implementation in SciPy of this general difference equation filter is a little more complicated then would be implied by the previous equation. It is implemented so that only one signal needs to be delayed. The actual implementation equations are (assuming $a_0 = 1$).

$$\begin{aligned}
y[n] &= b_0 x[n] + z_0[n-1] \\
z_0[n] &= b_1 x[n] + z_1[n-1] - a_1 y[n] \\
z_1[n] &= b_2 x[n] + z_2[n-1] - a_2 y[n] \\
&\vdots \\
z_{K-2}[n] &= b_{K-1} x[n] + z_{K-1}[n-1] - a_{K-1} y[n] \\
z_{K-1}[n] &= b_K x[n] - a_K y[n],
\end{aligned}$$

where $K = \max(N, M)$. Note that $b_K = 0$ if $K > M$ and $a_K = 0$ if $K > N$. In this way, the output at time $n$ depends only on the input at time $n$ and the value of $z_0$ at the previous time. This can always be calculated as long as the $K$ values $z_0[n-1] \ldots z_{K-1}[n-1]$ are computed and stored at each time step.

The difference-equation filter is called using the command **signal.lfilter** in SciPy. This command takes as inputs the vector $b$, the vector, $a$, a signal $x$ and returns the vector $y$ (the same length as $x$) computed using

the equation given above. If $x$ is $N$-dimensional, then the filter is computed along the axis provided. If, desired, initial conditions providing the values of $z_0[-1]$ to $z_{K-1}[-1]$ can be provided or else it will be assumed that they are all zero. If initial conditions are provided, then the final conditions on the intermediate variables are also returned. These could be used, for example, to restart the calculation in the same state.

Sometimes it is more convenient to express the initial conditions in terms of the signals $x[n]$ and $y[n]$. In other words, perhaps you have the values of $x[-M]$ to $x[-1]$ and the values of $y[-N]$ to $y[-1]$ and would like to determine what values of $z_m[-1]$ should be delivered as initial conditions to the difference-equation filter. It is not difficult to show that for $0 \leq m < K$,

$$z_m[n] = \sum_{p=0}^{K-m-1} \left(b_{m+p+1}x[n-p] - a_{m+p+1}y[n-p]\right).$$

Using this formula we can find the intial condition vector $z_0[-1]$ to $z_{K-1}[-1]$ given initial conditions on $y$ (and $x$). The command **signal.lfiltic** performs this function.

6.7.2.3. *Other filters.* The signal processing package provides many more filters as well.

Median Filter. A median filter is commonly applied when noise is markedly non-Gaussian or when it is desired to preserve edges. The median filter works by sorting all of the array pixel values in a rectangular region surrounding the point of interest. The sample median of this list of neighborhood pixel values is used as the value for the output array. The sample median is the middle array value in a sorted list of neighborhood values. If there are an even number of elements in the neighborhood, then the average of the middle two values is used as the median. A general purpose median filter that works on N-dimensional arrays is **signal.medfilt**. A specialized version that works only for two-dimensional arrays is available as **signal.medfilt2d**.

Order Filter. A median filter is a specific example of a more general class of filters called order filters. To compute the output at a particular pixel, all order filters use the array values in a region surrounding that pixel. These array values are sorted and then one of them is selected as the output value. For the median filter, the sample median of the list of array values is used as the output. A general order filter allows the user to select which of the sorted values will be used as the output. So, for example one could choose to pick the maximum in the list or the minimum. The order filter takes an additional argument besides the input array and the region mask that specifies which of the elements in the sorted list of neighbor array values should be used as the output. The command to perform an order filter is **signal.order_filter**.

Wiener filter. The Wiener filter is a simple deblurring filter for denoising images. This is not the Wiener filter commonly described in image reconstruction problems but instead it is a simple, local-mean filter. Let $x$ be the input signal, then the output is

$$y = \begin{cases} \frac{\sigma^2}{\sigma_x^2} m_x + \left(1 - \frac{\sigma^2}{\sigma_x^2}\right) x & \sigma_x^2 \geq \sigma^2, \\ m_x & \sigma_x^2 < \sigma^2. \end{cases}$$

Where $m_x$ is the local estimate of the mean and $\sigma_x^2$ is the local estimate of the variance. The window for these estimates is an optional input parameter (default is $3 \times 3$). The parameter $\sigma^2$ is a threshold noise parameter. If $\sigma$ is not given then it is estimated as the average of the local variances.

Hilbert filter. The Hilbert transform constructs the complex-valued analytic signal from a real signal. For example if $x = \cos \omega n$ then $y = \text{hilbert}(x)$ would return (except near the edges) $y = \exp(j\omega n)$. In the frequency domain, the hilbert transform performs

$$Y = X \cdot H$$

where $H$ is 2 for positive frequencies, 0 for negative frequencies and 1 for zero-frequencies.

Detrend.

### 6.7.3. Filter design.

6.7.3.1. *Finite-impulse response design.*
6.7.3.2. *Inifinite-impulse response design.*
6.7.3.3. *Analog filter frequency response.*
6.7.3.4. *Digital filter frequency response.*

### 6.7.4. Linear Time-Invariant Systems.

6.7.4.1. *LTI Object.*
6.7.4.2. *Continuous-Time Simulation.*
6.7.4.3. *Step response.*
6.7.4.4. *Impulse response.*

## 6.8. Input/Output

**6.8.1. Binary.**

6.8.1.1. *Arbitrary binary input and output (fopen).*

6.8.1.2. *Read and write Matlab .mat files.*

6.8.1.3. *Saving workspace.*

**6.8.2. Text-file.**

6.8.2.1. *Read text-files (read_array).*

6.8.2.2. *Write a text-file (write_array).*

## 6.9. Fourier Transforms

**6.9.1. One-dimensional.**

**6.9.2. Two-dimensional.**

**6.9.3. N-dimensional.**

**6.9.4. Shifting.**

**6.9.5. Sample frequencies.**

**6.9.6. Hilbert transform.**

**6.9.7. Tilbert transform.**

## 6.10. Linear Algebra

When SciPy is built using the optimized ATLAS LAPACK and BLAS libraries, it has very fast linear algebra capabilities. If you dig deep enough, all of the raw lapack and blas libraries are available for your use for even more speed. In this section, some easier-to-use interfaces to these routines are described.

All of these linear algebra routines expect an object that can be converted into a 2-dimensional array. The output of these routines is also a two-dimensional array. There is a matrix class defined in Numeric that scipy inherits and extends. You can initialize this class with an appropriate Numeric array in order to get objects for which multiplication is matrix-multiplication instead of the default, element-by-element multiplication.

**6.10.1. Matrix Class.** The matrix class is initialized with the SciPy command `mat` which is just convenient short-hand for `Matrix.Matrix`. If you are going to be doing a lot of matrix-math, it is convenient to convert arrays into matrices using this command. One convencience of using the mat command is that you can enter two-dimensional matrices using MATLAB-like syntax with commas or spaces separating columns and semicolons separting rows as long as the matrix is placed in a string passed to **mat**.

**6.10.2. Basic routines.**

6.10.2.1. *Finding Inverse.* The inverse of a matrix $\mathbf{A}$ is the matrix $\mathbf{B}$ such that $\mathbf{AB} = \mathbf{I}$ where $\mathbf{I}$ is the identity matrix consisting of ones down the main diagonal. Usually $\mathbf{B}$ is denoted $\mathbf{B} = \mathbf{A}^{-1}$. In SciPy, the matrix inverse of the Numeric array, A, is obtained using `linalg.inv(A)`, or using `A.I` if `A` is a Matrix. For example, let

$$\mathbf{A} = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 5 & 1 \\ 2 & 3 & 8 \end{bmatrix}$$

then

$$\mathbf{A^{-1}} = \frac{\mathbf{1}}{\mathbf{25}} \begin{bmatrix} -37 & 9 & 22 \\ 14 & 2 & -9 \\ 4 & -3 & 1 \end{bmatrix} = \begin{bmatrix} -1.48 & 0.36 & 0.88 \\ 0.56 & 0.08 & -0.36 \\ 0.16 & -0.12 & 0.04 \end{bmatrix}.$$

The following example demonstrates this computation in SciPy

```
>>> A = mat('[1 3 5; 2 5 1; 2 3 8]')
>>> A
Matrix([[1, 3, 5],
       [2, 5, 1],
       [2, 3, 8]])
>>> A.I
Matrix([[-1.48,  0.36,  0.88],
       [ 0.56,  0.08, -0.36],
       [ 0.16, -0.12,  0.04]])
```

```
>>> linalg.inv(A)
array([[-1.48,  0.36,  0.88],
       [ 0.56,  0.08, -0.36],
       [ 0.16, -0.12,  0.04]])
```

6.10.2.2. *Solving linear system.* Solving linear systems of equations is straightforward using the scipy command **linalg.solve.** This command expects an input matrix and a right-hand-side vector. The solution vector is then computed. An option for entering a symmetrix matrix is offered which can speed up the processing when applicable. As an example, suppose it is desired to solve the following simultaneous equations:

$$\begin{aligned} x + 3y + 5z &= 10 \\ 2x + 5y + z &= 8 \\ 2x + 3y + 8z &= 3 \end{aligned}$$

We could find the solution vector using a matrix inverse:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 5 & 1 \\ 2 & 3 & 8 \end{bmatrix}^{-1} \begin{bmatrix} 10 \\ 8 \\ 3 \end{bmatrix} = \frac{1}{25} \begin{bmatrix} -232 \\ 129 \\ 19 \end{bmatrix} = \begin{bmatrix} -9.28 \\ 5.16 \\ 0.76 \end{bmatrix}.$$

However, it is better to use the linalg.solve command which can be faster and more numerically stable. In this case it gives the same answer as shown in the following example:

```
>>> A = mat('[1 3 5; 2 5 1; 2 3 8]')
>>> b = mat('[10;8;3]')
>>> A.I*b
Matrix([[-9.28],
       [ 5.16],
       [ 0.76]])
>>> linalg.solve(A,b)
array([[-9.28],
       [ 5.16],
       [ 0.76]])
```

6.10.2.3. *Finding Determinant.* The determinant of a square matrix $\mathbf{A}$ is often denoted $|\mathbf{A}|$ and is a quantity often used in linear algebra. Suppose $a_{ij}$ are the elements of the matrix $\mathbf{A}$ and let $M_{ij} = |\mathbf{A}_{ij}|$ be the determinant of the matrix left by removing the $i^{\text{th}}$ row and $j^{\text{th}}$column from $\mathbf{A}$. Then for any row $i$,

$$|\mathbf{A}| = \sum_j (-1)^{i+j} a_{ij} M_{ij}.$$

This is a recursive way to define the determinant where the base case is defined by accepting that the determinant of a $1 \times 1$ matrix is the only matrix element. In SciPy the determinant can be calculated with **linalg.det**. For example, the determinant of

$$\mathbf{A} = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 5 & 1 \\ 2 & 3 & 8 \end{bmatrix}$$

is

$$\begin{aligned} |\mathbf{A}| &= 1 \begin{vmatrix} 5 & 1 \\ 3 & 8 \end{vmatrix} - 3 \begin{vmatrix} 2 & 1 \\ 2 & 8 \end{vmatrix} + 5 \begin{vmatrix} 2 & 5 \\ 2 & 3 \end{vmatrix} \\ &= 1\,(5 \cdot 8 - 3 \cdot 1) - 3\,(2 \cdot 8 - 2 \cdot 1) + 5\,(2 \cdot 3 - 2 \cdot 5) = -25. \end{aligned}$$

In SciPy this is computed as shown in this example:

```
>>> A = mat('[1 3 5; 2 5 1; 2 3 8]')
>>> linalg.det(A)
-25.000000000000004
```

6.10.2.4. *Computing norms.* Matrix and vector norms can also be computed with SciPy. A wide range of norm definitions are available using different parameters to the order argument of **linalg.norm**. This function takes a rank-1 (vectors) or a rank-2 (matrices) array and an optional order argument (default is 2). Based on these inputs a vector or matrix norm of the requested order is computed.

For vector $\mathbf{x}$, the order parameter can be any real number including **inf** or **-inf**. The computed norm is

$$\|\mathbf{x}\| = \begin{cases} \max|x_i| & \text{ord} = \text{inf} \\ \min|x_i| & \text{ord} = -\text{inf} \\ \left(\sum_i |x_i|^{\text{ord}}\right)^{1/\text{ord}} & |\text{ord}| < \infty. \end{cases}$$

For matrix $\mathbf{A}$ the only valid values for norm are $\pm 2, \pm 1$, $\pm$inf, and 'fro' (or 'f') Thus,

$$\|\mathbf{A}\| = \begin{cases} \max_i \sum_j |a_{ij}| & \text{ord} = \text{inf} \\ \min_i \sum_j |a_{ij}| & \text{ord} = -\text{inf} \\ \max_j \sum_i |a_{ij}| & \text{ord} = 1 \\ \min_j \sum_i |a_{ij}| & \text{ord} = -1 \\ \max \sigma_i & \text{ord} = 2 \\ \min \sigma_i & \text{ord} = -2 \\ \sqrt{\text{trace}\left(\mathbf{A}^H\mathbf{A}\right)} & \text{ord} = \text{'fro'} \end{cases}$$

where $\sigma_i$ are the singular values of $\mathbf{A}$.

6.10.2.5. *Solving linear least-squares problems and pseudo-inverses.* Linear least-squares problems occur in many branches of applied mathematics. In this problem a set of linear scaling coefficients is sought that allow a model to fit data. In particular it is assumed that data $y_i$ is related to data $\mathbf{x}_i$ through a set of coefficients $c_j$ and model functions $f_j(\mathbf{x}_i)$ via the model

$$y_i = \sum_j c_j f_j(\mathbf{x}_i) + \epsilon_i$$

where $\epsilon_i$ represents uncertainty in the data. The strategy of least squares is to pick the coefficients $c_j$ to minimize

$$J(\mathbf{c}) = \sum_i \left| y_i - \sum_j c_j f_j(x_i) \right|^2.$$

Theoretically, a global minimum will occur when

$$\frac{\partial J}{\partial c_n^*} = 0 = \sum_i \left( y_i - \sum_j c_j f_j(x_i) \right) \left(-f_n^*(x_i)\right)$$

or

$$\begin{aligned} \sum_j c_j \sum_i f_j(x_i) f_n^*(x_i) &= \sum_i y_i f_n^*(x_i) \\ \mathbf{A}^H\mathbf{A}\mathbf{c} &= \mathbf{A}^H\mathbf{y} \end{aligned}$$

where

$$\{\mathbf{A}\}_{ij} = f_j(x_i).$$

When $\mathbf{A^H A}$ is invertible, then

$$\mathbf{c} = \left(\mathbf{A}^H\mathbf{A}\right)^{-1}\mathbf{A}^H\mathbf{y} = \mathbf{A}^\dagger\mathbf{y}$$

where $\mathbf{A}^\dagger$ is called the pseudo-inverse of $\mathbf{A}$. Notice that using this definition of $\mathbf{A}$ the model can be written

$$\mathbf{y} = \mathbf{A}\mathbf{c} + \boldsymbol{\epsilon}.$$

The command **linalg.lstsq** will solve the linear least squares problem for $\mathbf{c}$ given $\mathbf{A}$ and $\mathbf{y}$. In addition **linalg.pinv** or **linalg.pinv2** (uses a different method based on singular value decomposition) will find $\mathbf{A}^\dagger$ given $\mathbf{A}$.

The following example Fig. 6.10.1 demonstrate the use of **linalg.lstsq** and **linalg.pinv** for solving a data-fitting problem. The data shown below were generated using the model:

$$y_i = c_1 e^{-x_i} + c_2 x_i$$

where $x_i = 0.1i$ for $i = 1 \ldots 10$, $c_1 = 5$, and $c_2 = 4$. Noise is added to $y_i$ and the coefficients $c_1$ and $c_2$ are estimated using linear least squares.

```
c1,c2= 5.0,2.0
i = r_[1:11]
xi = 0.1*i
yi = c1*exp(-xi)+c2*xi
zi = yi + 0.05*max(yi)*randn(len(yi))

A = c_[exp(-xi)[:,NewAxis],xi[:,NewAxis]]
```

```
c,resid,rank,sigma = linalg.lstsq(A,zi)

xi2 = r_[0.1:1.0:100j]
yi2 = c[0]*exp(-xi2) + c[1]*xi2

xplt.plot(xi,zi,'x',xi2,yi2)
xplt.limits(0,1.1,3.0,5.5)
xplt.xlabel('x_i')
xplt.title('Data fitting with linalg.lstsq')
xplt.eps('lstsq_fit')
```

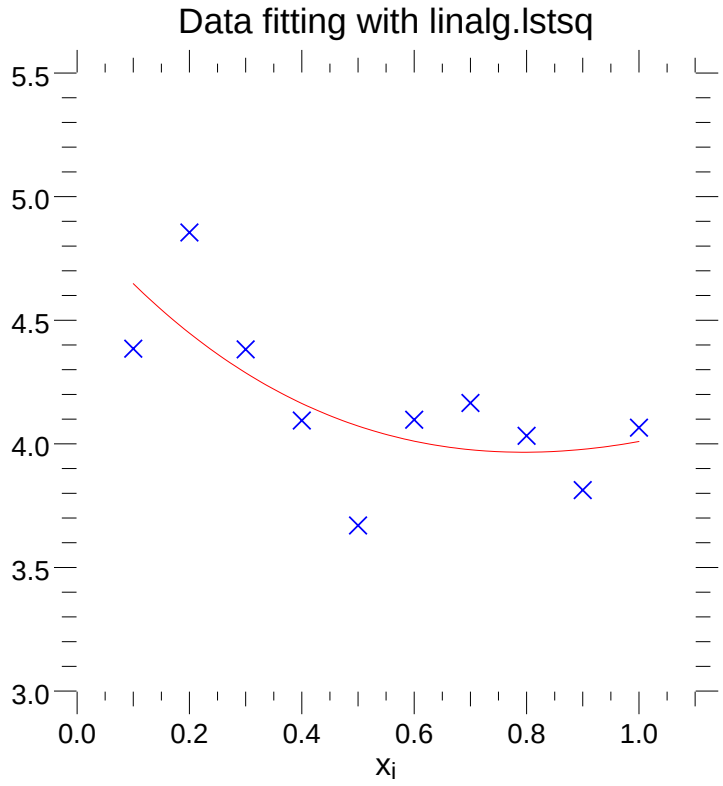



FIGURE 6.10.1. Least squares fitting example

6.10.2.6. *Generalized inverse.* The generalized inverse is calculated using the command **linalg.pinv** or **linalg.pinv2.** These two commands differ in how they compute the generalized inverse. The first uses the linalg.lstsq algorithm while the second uses singular value decomposition. Let $\mathbf{A}$ be an $M \times N$ matrix, then if $M > N$ the generalized inverse is

$$\mathbf{A}^{\dagger} = \left(\mathbf{A}^{H}\mathbf{A}\right)^{-1}\mathbf{A}^{H}$$

while if $M < N$ matrix the generalized inverse is

$$\mathbf{A}^{\#} = \mathbf{A}^{H}\left(\mathbf{A}\mathbf{A}^{H}\right)^{-1}.$$

In both cases for $M = N$, then

$$\mathbf{A}^{\dagger} = \mathbf{A}^{\#} = \mathbf{A}^{-1}$$

as long as $\mathbf{A}$ is invertible.

**6.10.3. Decompositions.** In many applications it is useful to decompose a matrix using other representations. There are several decompositions supported by SciPy.

6.10.3.1. *Eigenvalues and eigenvectors.* The eigenvalue-eigenvector problem is one of the most commonly employed linear algebra operations. In one popular form, the eigenvalue-eigenvector problem is to find for some square matrix $\mathbf{A}$ scalars $\lambda$ and corresponding vectors $\mathbf{v}$ such that

$$\mathbf{A}\mathbf{v} = \lambda\mathbf{v}.$$

For an $N \times N$ matrix, there are $N$ (not necessarily distinct) eigenvalues — roots of the (characteristic) polynomial

$$|\mathbf{A} - \lambda\mathbf{I}| = 0.$$

The eigenvectors, $\mathbf{v}$, are also sometimes called right eigenvectors to distinguish them from another set of left eigenvectors that satisfy

$$\mathbf{v}_L^H\mathbf{A} = \lambda\mathbf{v}_L^H$$

or

$$\mathbf{A}^H\mathbf{v}_L = \lambda^*\mathbf{v}_L.$$

With it's default optional arguments, the command **linalg.eig** returns $\lambda$ and $\mathbf{v}$. However, it can also return $\mathbf{v}_L$ and just $\lambda$ by itself (**linalg.eigvals** returns just $\lambda$ as well).

In addtion, **linalg.eig** can also solve the more general eigenvalue problem

$$\begin{aligned}\mathbf{A}\mathbf{v} &= \lambda\mathbf{B}\mathbf{v}\\ \mathbf{A}^H\mathbf{v}_L &= \lambda^*\mathbf{B}^H\mathbf{v}_L\end{aligned}$$

for square matrices $\mathbf{A}$ and $\mathbf{B}$. The standard eigenvalue problem is an example of the general eigenvalue problem for $\mathbf{B} = \mathbf{I}$. When a generalized eigenvalue problem can be solved, then it provides a decomposition of $\mathbf{A}$ as

$$\mathbf{A} = \mathbf{B}\mathbf{V}\mathbf{\Lambda}\mathbf{V}^{-1}$$

where $\mathbf{V}$ is the collection of eigenvectors into columns and $\mathbf{\Lambda}$ is a diagonal matrix of eigenvalues.

By definition, eigenvectors are only defined up to a constant scale factor. In SciPy, the scaling factor for the eigenvectors is chosen so that $\|\mathbf{v}\|^2 = \sum_i v_i^2 = 1$.

As an example, consider finding the eigenvalues and eigenvectors of the matrix

$$\mathbf{A} = \begin{bmatrix} 1 & 5 & 2 \\ 2 & 4 & 1 \\ 3 & 6 & 2 \end{bmatrix}.$$

The characteristic polynomial is

$$\begin{aligned}|\mathbf{A} - \lambda\mathbf{I}| &= (1-\lambda)\left[(4-\lambda)(2-\lambda) - 6\right] - \\ &\quad 5\left[2(2-\lambda) - 3\right] + 2\left[12 - 3(4-\lambda)\right] \\ &= -\lambda^3 + 7\lambda^2 + 8\lambda - 3.\end{aligned}$$

The roots of this polynomial are the eigenvalues of $\mathbf{A}$:

$$\begin{aligned}\lambda_1 &= 7.9579\\ \lambda_2 &= -1.2577\\ \lambda_3 &= 0.2997.\end{aligned}$$

The eigenvectors corresponding to each eigenvalue can be found using the original equation. The eigenvectors associated with these eigenvalues can then be found.

```
>>> A = mat('[1 5 2; 2 4 1; 3 6 2]')
>>> la,v = linalg.eig(A)
>>> l1,l2,l3 = la
>>> print l1, l2, l3
(7.95791620491+0j) (-1.25766470568+0j) (0.299748500767+0j)

>>> print v[:,0]
array([-0.5297, -0.4494, -0.7193])
>>> print v[:,1]
[-0.9073  0.2866  0.3076]
>>> print v[:,2]
[ 0.2838 -0.3901  0.8759]
>>> print sum(abs(v**2),axis=0)
[ 1.  1.  1.]

>>> v1 = mat(v[:,0]).T
```

```
>>> print max(ravel(abs(A*v1-l1*v1)))
4.4408920985e-16
```

6.10.3.2. *Singular value decomposition.* Singular Value Decompostion (SVD) can be thought of as an extension of the eigenvalue problem to matrices that are not square. Let $\mathbf{A}$ be an $M \times N$ matrix with $M$ and $N$ arbitrary. The matrices $\mathbf{A}^H\mathbf{A}$ and $\mathbf{A}\mathbf{A}^H$ are square hermitian matrices[3] of size $N \times N$ and $M \times M$ respectively. It is known that the eigenvalues of square hermitian matrices are real and non-negative. In addtion, there are at most $\min(M, N)$ identical non-zero eigenvalues of $\mathbf{A}^H\mathbf{A}$ and $\mathbf{A}\mathbf{A}^H$. Define these positive eigenvalues as $\sigma_i^2$. The square-root of these are called singular values of $\mathbf{A}$. The eigenvectors of $\mathbf{A}^H\mathbf{A}$ are collected by columns into an $N \times N$ unitary[4] matrix $\mathbf{V}$ while the eigenvectors of $\mathbf{A}\mathbf{A}^H$ are collected by columns in the unitary matrix $\mathbf{U}$, the singular values are collected in an $M \times N$ zero matrix $\mathbf{\Sigma}$ with main diagonal entries set to the singular values. Then

$$\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^H$$

is the singular-value decomposition of $\mathbf{A}$. Every matrix has a singular value decomposition. Sometimes, the singular values are called the spectrum of $\mathbf{A}$. The command **linalg.svd** will return $\mathbf{U}$, $\mathbf{V}^H$, and $\sigma_i$ as an array of the singular values. To obtain the matrix $\mathbf{\Sigma}$ use **linalg.diagsvd.** The following example illustrates the use of **linalg.svd**.

```
>>> A = mat('[1 3 2; 1 2 3]')
>>> M,N = A.shape
>>> U,s,Vh = linalg.svd(A)
>>> Sig = mat(diagsvd(s,M,N))
>>> U, Vh = mat(U), mat(Vh)
>>> print U
Matrix([[-0.7071, -0.7071],
       [-0.7071,  0.7071]])
>>> print Sig
Matrix([[ 5.1962,  0.    ,  0.    ],
       [ 0.    ,  1.    ,  0.    ]])
>>> print Vh
Matrix([[-0.2722, -0.6804, -0.6804],
       [-0.    , -0.7071,  0.7071],
       [-0.9623,  0.1925,  0.1925]])

>>> print A
Matrix([[1, 3, 2],
       [1, 2, 3]])
>>> print U*Sig*Vh
Matrix([[ 1.,  3.,  2.],
       [ 1.,  2.,  3.]])
```

6.10.3.3. *LU decomposition.* The LU decompostion finds a representation for the $M \times N$ matrix $\mathbf{A}$ as

$$\mathbf{A} = \mathbf{P}\mathbf{L}\mathbf{U}$$

where $\mathbf{P}$ is an $M \times M$ permutation matrix (a permutation of the rows of the identity matrix), $\mathbf{L}$ is in $M \times K$ lower triangular or trapezoidal matrix ($K = \min(M, N)$) with unit-diagonal, and $\mathbf{U}$ is an upper triangular or trapezoidal matrix. The SciPy command for this decomposition is **linalg.lu**.

Such a decomposition is often useful for solving many simultaneous equations where the left-hand-side does not change but the right hand side does. For example, suppose we are going to solve

$$\mathbf{A}\mathbf{x}_i = \mathbf{b}_i$$

for many different $\mathbf{b}_i$. The LU decomposition allows this to be written as

$$\mathbf{P}\mathbf{L}\mathbf{U}\mathbf{x}_i = \mathbf{b}_i.$$

Because $\mathbf{L}$ is lower-triangular, the equation can be solved for $\mathbf{U}\mathbf{x}_i$ and finally $\mathbf{x}_i$ very rapidly using forward- and back-substitution. An initial time spent factoring $\mathbf{A}$ allows for very rapid solution of similar systems of equations in the future. If the intent for performing LU decomposition is for solving linear systems then the command **linalg.lu_factor** should be used followed by repeated applications of the command **linalg.lu_solve** to solve the system for each new right-hand-side.

[3] A hermition matrix $\mathbf{D}$ satisfies $\mathbf{D}^H = \mathbf{D}$.

[4] A unitary matrix $\mathbf{D}$ satisfies $\mathbf{D}^H\mathbf{D} = \mathbf{I} = \mathbf{D}\mathbf{D}^H$ so that $\mathbf{D}^{-1} = \mathbf{D}^H$.

6.10.3.4. *Cholesky decomposition.* Cholesky decomposition is a special case of LU decomposition applicable to Hermitian positive definite matrices. When $\mathbf{A} = \mathbf{A}^H$ and $\mathbf{x}^H\mathbf{A}\mathbf{x} \geq 0$ for all $\mathbf{x}$, then decompositions of $\mathbf{A}$ can be found so that

$$\begin{aligned}\mathbf{A} &= \mathbf{U}^H\mathbf{U} \\ \mathbf{A} &= \mathbf{L}\mathbf{L}^H\end{aligned}$$

where $\mathbf{L}$ is lower-triangular and $\mathbf{U}$ is upper triangular. Notice that $\mathbf{L} = \mathbf{U}^H$. The command **linagl.cholesky** computes the cholesky factorization. For using cholesky factorization to solve systems of equations there are also **linalg.cho_factor** and **linalg.cho_solve** routines that work similarly to their LU decomposition counterparts.

6.10.3.5. *QR decomposition.* The QR decomposition (sometimes called a polar decomposition) works for any $M \times N$ array and finds an $M \times M$ unitary matrix $\mathbf{Q}$ and an $M \times N$ upper-trapezoidal matrix $\mathbf{R}$ such that

$$\mathbf{A} = \mathbf{Q}\mathbf{R}.$$

Notice that if the SVD of $\mathbf{A}$ is known then the QR decomposition can be found

$$\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^H = \mathbf{Q}\mathbf{R}$$

implies that $\mathbf{Q} = \mathbf{U}$ and $\mathbf{R} = \mathbf{\Sigma}\mathbf{V}^H$. Note, however, that in SciPy independent algorithms are used to find QR and SVD decompositions. The command for QR decomposition is **linalg.qr**.

6.10.3.6. *Schur decomposition.* For a square $N \times N$ matrix, $\mathbf{A}$, the Schur decomposition finds (not-necessarily unique) matrices $\mathbf{T}$ and $\mathbf{Z}$ such that

$$\mathbf{A} = \mathbf{Z}\mathbf{T}\mathbf{Z}^H$$

where $\mathbf{Z}$ is a unitary matrix and $\mathbf{T}$ is either upper-triangular or quasi-upper triangular depending on whether or not a real schur form or complex schur form is requested. For a real schur form both $\mathbf{T}$ and $\mathbf{Z}$ are real-valued when $\mathbf{A}$ is real-valued. When $\mathbf{A}$ is a real-valued matrix the real schur form is only quasi-upper triangular because $2 \times 2$ blocks extrude from the main diagonal corresponding to any complex-valued eigenvalues. The command **linalg.schur** finds the Schur decomposition while the command **linalg.rsf2csf** converts $\mathbf{T}$ and $\mathbf{Z}$ from a real Schur form to a complex Schur form. The Schur form is especially useful in calculating functions of matrices.

The following example illustrates the schur decomposition:

```
>>> A = mat('[1 3 2; 1 4 5; 2 3 6]')
>>> T,Z = linalg.schur(A)
>>> T1,Z1 = linalg.schur(A,'complex')
>>> T2,Z2 = linalg.rsf2csf(T,Z)
>>> print T
Matrix([[ 9.9001,  1.7895, -0.655 ],
       [ 0.    ,  0.5499, -1.5775],
       [ 0.    ,  0.5126,  0.5499]])
>>> print T2
Matrix([[ 9.9001+0.j    , -0.3244+1.5546j, -0.8862+0.569j ],
       [ 0.    +0.j    ,  0.5499+0.8993j,  1.0649-0.j    ],
       [ 0.    +0.j    ,  0.    +0.j    ,  0.5499-0.8993j]])
>>> print abs(T1-T2) # different
[[ 0.      2.1184  0.1949]
 [ 0.      0.      1.2676]
 [ 0.      0.      0.    ]]
>>> print abs(Z1-Z2) # different
[[ 0.0683  1.1175  0.1973]
 [ 0.1186  0.5644  0.247 ]
 [ 0.1262  0.7645  0.1916]]
>>> T,Z,T1,Z1,T2,Z2 = map(mat,(T,Z,T1,Z1,T2,Z2))
>>> print abs(A-Z*T*Z.H)
Matrix([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
>>> print abs(A-Z1*T1*Z1.H)
Matrix([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
>>> print abs(A-Z2*T2*Z2.H)
Matrix([[ 0.,  0.,  0.],
```

```
[ 0.,  0.,  0.],
[ 0.,  0.,  0.]])
```

**6.10.4. Matrix Functions.** Consider the function $f(x)$ with Taylor series expansion

$$f(x) = \sum_{k=0}^{\infty} \frac{f^{(k)}(0)}{k!} x^k.$$

A matrix function can be defined using this Taylor series for the square matrix $\mathbf{A}$ as

$$f(\mathbf{A}) = \sum_{k=0}^{\infty} \frac{f^{(k)}(0)}{k!} \mathbf{A}^k.$$

While, this serves as a useful representation of a matrix function, it is rarely the best way to calculate a matrix function.

6.10.4.1. *Exponential and logarithm functions.* The matrix exponential is one of the more common matrix functions. It can be defined for square matrices as

$$e^{\mathbf{A}} = \sum_{k=0}^{\infty} \frac{1}{k!} \mathbf{A}^k.$$

The command **linalg.expm3** uses this Taylor series definition to compute the matrix exponential. Due to poor convergence properties it is not often used.

Another method to compute the matrix exponential is to find an eigenvalue decomposition of $\mathbf{A}$:

$$\mathbf{A} = \mathbf{V}\mathbf{\Lambda}\mathbf{V}^{-1}$$

and note that

$$e^{\mathbf{A}} = \mathbf{V}e^{\mathbf{\Lambda}}\mathbf{V}^{-1}$$

where the matrix exponential of the diagonal matrix $\mathbf{\Lambda}$ is just the exponential of its elements. This method is implemented in **linalg.expm2**.

The preferred method for implementing the matrix exponential is to use scaling and a PadÃ© approximation for $e^x$. This algorithm is implemented as **linalg.expm**.

The inverse of the matrix exponential is the matrix logarithm defined as the inverse of the matrix exponential.

$$\mathbf{A} \equiv \exp(\log(\mathbf{A})).$$

The matrix logarithm can be obtained with **linalg.logm**.

6.10.4.2. *Trigonometric functions.* The trigonometric functions sin, cos, and tan are implemented for matrices in **linalg.sinm, linalg.cosm,** and **linalg.tanm** respectively. The matrix sin and cosine can be defined using Euler's identity as

$$\begin{aligned} \sin(\mathbf{A}) &= \frac{e^{j\mathbf{A}} - e^{-j\mathbf{A}}}{2j} \\ \cos(\mathbf{A}) &= \frac{e^{j\mathbf{A}} + e^{-j\mathbf{A}}}{2}. \end{aligned}$$

The tangent is

$$\tan(x) = \frac{\sin(x)}{\cos(x)} = [\cos(x)]^{-1} \sin(x)$$

and so the matrix tangent is defined as

$$[\cos(\mathbf{A})]^{-1} \sin(\mathbf{A}).$$

6.10.4.3. *Hyperbolic trigonometric functions.* The hyperbolic trigonemetric functions sinh, cosh, and tanh can also be defined for matrices using the familiar definitions:

$$\begin{aligned} \sinh(\mathbf{A}) &= \frac{e^{\mathbf{A}} - e^{-\mathbf{A}}}{2} \\ \cosh(\mathbf{A}) &= \frac{e^{\mathbf{A}} + e^{-\mathbf{A}}}{2} \\ \tanh(\mathbf{A}) &= [\cosh(\mathbf{A})]^{-1} \sinh(\mathbf{A}). \end{aligned}$$

These matrix functions can be found using **linalg.sinhm**, **linalg.coshm**, and **linalg.tanhm**.

6.10.4.4. *Arbitrary function.* Finally, any arbitrary function that takes one complex number and returns a complex number can be called as a matrix function using the command **linalg.funm**. This command takes the matrix and an arbitrary Python function. It then implements an algorithm from Golub and Van Loan's book "Matrix Computations" to compute function applied to the matrix using a Schur decomposition. Note that *the function needs to accept complex numbers* as input in order to work with this algorithm. For example the following code computes the zeroth-order Bessel function applied to a matrix.

```
>>> A = rand(3,3)
>>> B = linalg.funm(A,lambda x: special.jv(0,real(x)))
>>> print A
[[ 0.0593  0.5612  0.4403]
 [ 0.8797  0.2556  0.1452]
 [ 0.964   0.9666  0.1243]]
>>> print B
[[ 0.8206 -0.1212 -0.0612]
 [-0.1323  0.8256 -0.0627]
 [-0.2073 -0.1946  0.8516]]
```

## 6.11. Statistics

SciPy has a tremendous number of basic statistics routines with more easily added by the end user (if you create one please contribute it). All of the statistics functions are located in the sub-package **stats** and a fairly complete listing of these functions can be had using `info(stats)`.

**6.11.1. Random Variables.** There are two general distribution classes that have been implemented for encapsulating continuous random variables and discrete random variables. Over 80 continuous random variables and 10 discrete random variables have been implemented using these classes. The list of the random variables available is in the docstring for the stats sub-package. A detailed description of each of them is also located in the files continuous.lyx and discrete.lyx in the stats sub-directories.

## 6.12. Interfacing with the Python Imaging Library

If you have the Python Imaging Library (PIL) installed, SciPy provides some convenient functions that make use of it's facilities particularly for reading, writing, displaying, and rotating images. In SciPy an image is always a two- or three-dimensional array. Gray-scale, and colormap images are always two-dimensional arrays while RGB images are three-dimensional with the third dimension specifying the channel.

Commands available include

- fromimage — convert a PIL image to a Numeric array
- toimage — convert Numeric array to PIL image
- imsave — save Numeric array to an image file
- imread — read an image file into a Numeric array
- imrotate — rotate an image (a Numeric array) counter-clockwise
- imresize — resize an image using the PIL
- imshow — external viewer of a Numeric array (using PIL)
- imfilter — fast, simple built-in filters provided by PIL
- radon — simple radon transform based on imrotate

## 6.13. Some examples

```
"""Simple data fitting and smoothing example"""

from scipy import exp, arange,array, linspace
from RandomArray import normal
from scipy.optimize import leastsq
from scipy.interpolate import splrep,splev

import pylab as P

parsTrue = array([2.0, -.76, 0.1])
distance = linspace(0, 4, 1000)
```

```
def func(pars):
    a, alpha, k = pars
    return a*exp(alpha*distance) + k

def errfunc(pars):
    return data - func(pars)  #return the error

# some pseudo data; add some noise
data = func(parsTrue) + normal(0.0, 0.1, distance.shape)

# the intial guess of the params
guess = 1.0, -.4, 0.0

# now solve for the best fit paramters
best, info, ier, mesg = leastsq(errfunc, guess, full_output=1)

print 'true', parsTrue
print 'best', best
print '|err|_l2 =',P.l2norm(parsTrue-best)

# scipy's splrep uses FITPACK's curfit (B-spline interpolation)
print 'Spline smoothing of the data'
sp = splrep(distance,data)
smooth = splev(distance,sp)
print 'Spline information (see splrep and splev for details):',sp

# Now use pylab to plot
P.figure()
P.plot(distance,data,label='Noisy data')
P.plot(distance,func(best),lw=2,label='Best fit')
P.legend()
P.figure()
P.plot(distance,data,label='Noisy data')
P.plot(distance,smooth,lw=2,label='Spline-smoothing')
P.legend()
P.show()


import scipy as S
import pylab as P

# poly1d objects are constructed from coefficients, highest-order first.
coefs = [  1. ,  -3.5,  -0.5,  11. , -17. ,   6. ]

pol = S.poly1d(coefs)
roots = pol.r

print 'Polynomial p(x):\n',pol,'\n'
print 'p(x) built with coefs:',coefs
print 'Roots of p(x):',roots

# Plot p(x)
x = P.frange(-4,4,npts=400)
y = pol(x)
```

```
P.figure()
P.axhline(0,color='g')
P.axvline(0,color='g')
P.plot(x,y,'b-')

# Show roots
P.scatter(roots.real,roots.imag,s=80,c='r')

# Set limits and grid to make the plot clear
P.ylim(-40,40)
P.grid()

# Display on screen
P.show()


"""Plot some Bessel functions of integer order, using Scipy and pylab"""

import scipy as S
import pylab as P

# shorthand
special = S.special

def jn_asym(n,x):
    """Asymptotic form of jn(x) for x>>n"""

    return S.sqrt(2.0/S.pi/x)*S.cos(x-(n*S.pi/2.0+S.pi/4.0))

# build a range of values to plot in
x = P.frange(0,30,npts=400)

# Start by plotting the well-known j0 and j1
P.figure()
P.plot(x,special.j0(x),label='j0')
P.plot(x,special.j1(x),label='j1')

# Show a higher-order Bessel function
n = 5
P.plot(x,special.jn(n,x),label='j%s' % n)

# and compute its asymptotic form (valid for x>>n, where n is the order).  We
# must first find the valid range of x where at least x>n:
x_asym = S.compress(x>n,x)
P.plot(x_asym,jn_asym(n,x_asym),label='j%s (asymptotic)' % n)

# Finish off the plot
P.legend()
P.title('Bessel Functions')
# horizontal line at 0 to show x-axis, but after the legend
P.axhline(0)

# EXERCISE: redo the above, for the asymptotic range 0<x<<n.  The asymptotic
# form in this regime is
```

```
# J(n,x) = (1/gamma(n+1))(x/2)^n

# Now, let's verify numerically the recursion relation
# J(n+1,x) = (2n/x)J(n,x)-J(n-1,x)
jn = special.jn  # just a shorthand

# Be careful to check only for x!=0, to avoid divisions by zero
xp = S.compress(x>0.0,x)  # positive x

# construct both sides of the recursion relation, these should be equal
j_np1 = jn(n+1,xp)
j_np1_rec = (2.0*n/xp)*jn(n,xp)-jn(n-1,xp)

# Now make a nice error plot of the difference, in a new figure
P.figure()
P.semilogy(xp,abs(j_np1-j_np1_rec),'r+-')
P.title('Error in recursion for J%s' % n)
P.grid()

# Don't forget a show() call at the end of the script
P.show()
```

CHAPTER 7

# 3D visualization with MayaVi

Chapter contributed by Prabhu Ramachandran

## 7.1. Introduction

MayaVi is a scientific data visualizer. It is written in Python <http://www.python.org> and uses the Visualization Toolkit (VTK) <http://www.vtk.org/> for the visualization. An easy to use GUI using Tkinter <http://www.pythonware.com/library/tkinter/introduction/index.htm> is provided. MayaVi is free software and is distributed under the conditions of the BSD license <http://www.opensource.org/licenses/bsd-license.html>. It is also cross platform and should run on any platform where both Python and VTK are available (which is almost any *nix, Mac OSX or Windows).

In Sanskrit "Mayavi" means magician. The name wasn't exactly chosen for its meaning but was the result of a long and hard search with the author pestering a lot of people for suggestions. My sincere thanks to all of those who offered suggestions.

MayaVi has a quite a few useful features:

- An easy to use GUI.
- MayaVi can be used as a Python module from other Python programs. MayaVi can also be used interactively from the Python interpreter.
- Provides modules to visualize grids, scalar and vector data fields. Rudimentary tensor support is also available.
- It provides support for *any* VTK dataset using the VTK data format <http://www.vtk.org/pdf/file-formats.pdf>. This includes rectilinear, structured and unstructured grid data and also polygonal data. Both the original VTK data formats and the new XML formats are supported.
- Support for PLOT3D data. Both ASCII and binary files work. Only the structured grid format works because of current limitations in the vtkPLOT3DReader. Simple support for multi-block data is also incorporated.
- Support for EnSight data. EnSight6 and EnSightGold formats are supported. Only single parts are supported at this time.
- Many datasets can be used simultaneously.
- Multiple visualization modules can be used simultaneously.
- Quite a few basic data filters are also provided.
- Supports volume visualization of data via texture and ray cast mappers.
- Support for importing a simple VRML scene or a 3D Studio file. Texturing is not yet supported due to limitations in VTK's vtkVRMLImporter.
- A pipeline browser with which one can browse and edit objects in the VTK pipeline. A segmented pipeline browser is used to make it easier to look at parts of the VTK pipeline.
- A modular design so one can add ones own modules and filters.
- A Lookup Table editor to customize lookup tables easily while visualizing data!
- The visualization (or a part of it) can be saved and reloaded in the future.
- Export the visualized scene to a Post Script file, PPM/BMP/TIFF/JPEG/PNG image, Open Inventor, Geomview OOGL, VRML files or RenderMan RIB files. It is also possible to save the scene to a vector graphic via GL2PS <http://www.geuz.org/gl2ps>. This is only available if VTK is built with GL2PS support.
- Support for picking data points or cells and also configuring the lighting of the visualization.

And a lot more. MayaVi is free software and hence can be modified to do things differently. The rest of this manual will provide information on how to use it.

## 7.2. Getting started

MayaVi is a pretty powerful tool. This chapter describes the GUI that MayaVi provides and the way to use it. This chapter gets you started using MayaVi.

**7.2.1. Starting MayaVi.** Under *nix if your installation is setup such that the script **mayavi** is on the system wide path just run the executable **mayavi** anywhere. If not, change the current directory to the directory where MayaVi was installed and run:

If you have an already saved MayaVi visualization in some file, say `saved_viz.mv` you can start MayaVi using that file like so:

Under Windows visit the directory where MayaVi was installed and double click on the executable **mayavi.pyw**. If your installation went well this should start MayaVi.

If you have problems running MayaVi, consult the MayaVi home page and look at the Installation sections. You can also ask for help at the mailing list or ask the author.

**7.2.2. Command line arguments.**

7.2.2.1. *Basic options.* This section lists some simple useful command line options

`mayavi --display DISPLAY:` Use DISPLAY for the X display. This option makes sense only when running MayaVi under X.

`mayavi -g WIDTHxHEIGHT+XOFF+YOFF:` Set the geometry of the main window when it is launched. The arguments that can be passed follow the standard X convention and include the width, height, x offset and y offset of the window. This option is also available through `--geometry`.

`mayavi -h:` This prints all the available command line options and exits. Also available through `--help`.

`mayavi -V:` This prints the MayaVi version on the command line and exits. Also available through `--version`.

`mayavi filename.mv:` This loads a previously saved MayaVi visualization.

7.2.2.2. *Advanced options.* This section lists some advanced command line options. This section is intended for those who already understand how MayaVi works. If you are new to MayaVi it is recommended that you read the rest of the guide and then get back here when you need more advanced command line options.

`mayavi -d vtk_file.vtk:` Opens a VTK file (even the new XML format is supported) passed as the argument. Also available through `--vtk`.

`mayavi -x plot3d_xyz_file:` This opens a PLOT3D co-ordinate file passed as the argument. Also available through `--plot3d-xyz`.

`mayavi -q plot3d_q_file:` This opens a PLOT3D solution file passed as the argument. Please note that this option must *always* follow a -q or –plot3d-xyz option. Also available through `--plot3d-q`.

`mayavi -e ensight_case_file:` Opens an EnSight case file passed as the argument. Also available through `--ensight`.

`mayavi -m module-name:` The passed module name is loaded in the current ModuleManager. The module name must be a valid one if not you will get an error message. Also available through `--module`.

`mayavi -f filter-name:` The passed filter name is loaded in the current ModuleManager. The filter name must be a valid one if not you will get an error message. Also available through `--filter`. If the filter is the `UserDefined` filter then it could be specified as `UserDefined:vtkSomeFilter` where `vtkSomeFilter` is a valid VTK class. In this case the filter will not prompt you for the VTK filter to use.

`mayavi -z saved-visualization-file:` Loads a previously saved MayaVi visualization file passed as the argument. Also available through `--viz` and `--visualization`.

`mayavi -M module-manager-file:` Loads a module manager saved to a file. If a file that does not exist is given this will simply create a new module manager that can be populated with filters and modules. Also available through `--module-mgr`.

`mayavi -w vrml2-file:` Imports a VRML2 scene given an appropriate file. Also available through `--vrml`.

`mayavi -3 3DStudio-file:` Imports a 3D Studio scene given an appropriate file. Also available through `--3ds`.

`mayavi -n:` Creates a new window. Any options passed after this will apply to this newly created window. Also available through `--new-window`.

7.2.2.3. *Examples.* Here are a few interesting examples.

This command loads an existing visualization.

This command loads the `heart.mv` saved visualization in one window, creates a new window and loads the `other.mv` in the other.

```
>  -M new -f Threshold -m IsoSurface \
>  -n -d examples/heart.vtk -m Outline -m ContourGridPlane
```

This command loads a VTK data file called `heart.vtk`, loads the `Axes`, `GridPlane` modules in one `ModuleManager`. Then creates a new `ModuleManager` and loads a `Threshold` filter and an `IsoSurface` module in it. It then opens a new visualization window, loads the VTK data file, `heart.vtk`, the modules `Outline` and `ContourGridPlane` in it.

The provided options make it possible to construct very useful visualizations from the command line.

**7.2.3. The MayaVi Window.** MayaVi provides an easy to use GUI. The picture shown below shows the basic GUI that MayaVi provides. The regions marked out in red are to be noted. The top left shows a set of menus. Below the menus is a control panel on the left and the actual visualization on the right. At the bottom of the application window is a status bar that turns red when MayaVi is busy doing something. In between the status bar and the visualization are provided a set of buttons that help control the visualization view.
Each section of the screen marked and described above provides important functionality.

**Menu:**
This provides a set of menus from which provide the user with bulk of the functionality.

**Visualization:**
This part of the screen is where the data is visualized using VTK.

**Control Panel:**
The control panel allows the user to configure and control the particular visualization. It provides various lists for the user's convenience. These are discussed in detail subsequently.

**Status Bar:**
This part of the screen indicates the status of MayaVi to the user. If MayaVi is busy doing something this part of the screen will turn red and the cursor will change to a *watch* indicating that MayaVi is busy.

**View Modes:**
These are a set of convenience buttons that help the user quickly see one particular view of the visualization.

The next chapter deals with using MayaVi.

## 7.3. Using MayaVi

This chapter describes in detail the way to use MayaVi for your data visualization.

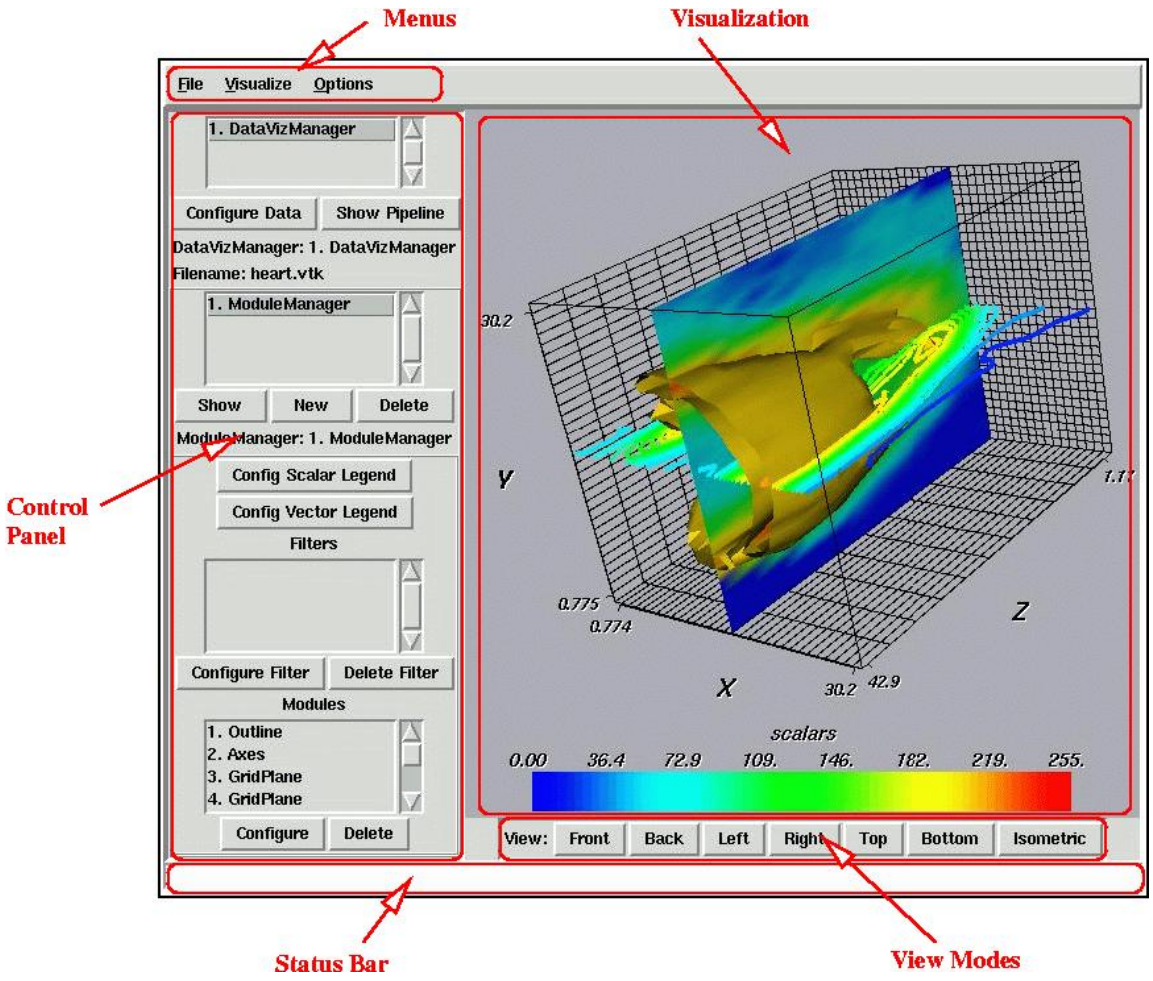



FIGURE 7.2.1. MayaVi window

**7.3.1. The Basic Design of MayaVi.** It is important to understand the basic design of MayaVi before you use it. MayaVi is a data visualizer and the design reflects this. The basic design is simple and is described in this section.

7.3.1.1. *The Control Panel.* The control panel needs to be understood before one can do anything serious with MayaVi. This section describes the control panel in some detail.

- Associated with each data file that is to be visualized is an object called a `DataVizManager`. This object is responsible for the datafile and the entire visualization associated with that data file. Each `DataVizManager` instance is shown in the first list in the control panel.
- Each `DataVizManager` controls a set of `ModuleManager`s. These `ModuleManager`s are listed in the second list from the top.
- Each `ModuleManager` controls set of two legends (one for scalar visualization and one for vector visualization), a collection of `Filter`s and a collection of `Module`s. Any number of Filters and Modules can be used.
- A `Filter` is an object that filters out the data in some way or the other. A simple example is the `ExtractVectorNorm` filter. This extracts the magnitude of the selected vector data field attribute for the data file. All modules that use this as an input will receive the vector magnitude as the scalar field data. The filters can be chosen from the **Visualize** menu. Each `ModuleManager` can have as many filters as are required. When multiple filters are used, it is important to note that each filter sends its data to the next filter in sequence. This could be problematic in some situations. Lets say there is a structured grid object and that needs to be subsampled. We can use the `ExtractGrid` filter and then display a `GridPlane`. Now we want to show contours but this time we want to threshold the

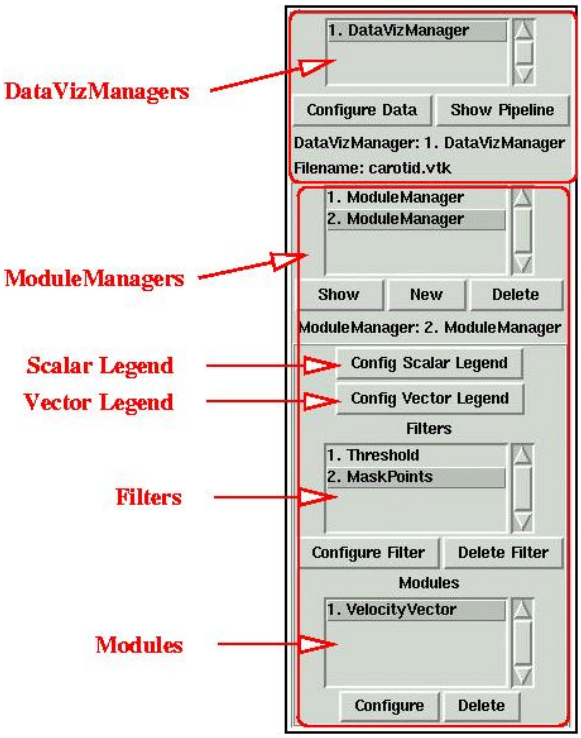



FIGURE 7.3.1. MayaVi control panel

contours based on input scalars so we use the `Threshold` filter. If we put the `Threshold` filter in the same ModuleManager, the grid will disappear since the `Threshold` output is an unstructured grid. So to handle this we create a new `ModuleManager` (click the **New** button) and add the `Threshold` filter in that `ModuleManager`. Put all the modules that use this filter in that `ModuleManager`.

- A `Module` is an object that actually visualizes the data. There are a large number of Modules that can be used and these are also available from the **Visualize** menu. Each `ModuleManager` can have as many modules as are required.

Although the above sounds complex, it really is not. It may just take a little getting used to before you are completely comfortable with it. The following figure illustrates the above and clarifies matters.

If you have multiple `DataVizManager`s and want to see the `ModuleManager`s of one of them then you either double click on the appropriate manager or single click on the manager and click on the **Show Pipeline** for the `DataVizManager`s and on the **Show** for the `ModuleManager`s.

The other GUI buttons and features are all rather self explanatory. There are only a few points that need to be made in order to make this description complete:

- The creation and deletion of a `DataVizManager` is controlled via the **File** menu. The open menu items will create a new `DataVizManager` and the **Close** menu item will close the selected `DataVizManager` and all its `ModuleManager`s. The **Close All** menu item will close all the `DataVizManager`s.
- The addition and deletion of `ModuleManager`s, can be done using the **New** and **Delete** buttons in the control panel.
- `Filter`s and `Module`s can be added from the **Visualize** menu and the corresponding **Filters** and **Modules** the sub menus. They can be deleted from the control panel.
- `Filter`s and `Module`s can be configured by either double clicking on the corresponding item or by selecting the item and clicking on the **configure** button.

**7.3.2. Data formats.** MayaVi is a data visualizer and one first needs to have data to visualize anything. MayaVi supports the following data formats:

**VTK data format** <`http://www.vtk.org/pdf/file-formats.pdf`>**:**

MayaVi supports *any* VTK dataset using the VTK data format <`http://www.vtk.org/pdf/file-formats.pdf`>. This includes rectilinear, structured and unstructured grid data and also polygonal data. Please refer the linked document for details on the VTK data format. The new VTK XML data format is also supported by MayaVi.

**PLOT3D data:**

MayaVi supports PLOT3D file format with binary structured grid data. The other PLOT3D datasets will not work due to limitations in VTK's vtkPLOT3DReader. Simple support for multi-block data is also incorporated.

**EnSight data:**

MayaVi supports EnSight data. EnSight6 and EnSightGold formats are supported. Only single parts are supported at this time.

In addition to this MayaVi allows one to import VRML2 files and 3D Studio files. Texturing is not yet supported due to limitations in VTK's vtkVRMLImporter.

Those interested in converting data arrays under Python into VTK files should look at Pearu Peterson's pyVTK <`http://cens.ioc.ee/projects/pyvtk/`> Python module.

MayaVi ships with a simple example of a heart CT scan data in the VTK data format (this should be in the `examples/` directory). The dataset used is a structured points dataset with a scalar field. This can be used as a simple reference. More VTK, PLOT3D and EnSight data samples should be available from the VTK download page <`http://www.vtk.org/get-software.php`>.

**7.3.3. Opening a data file.** Once data suitable for MayaVi is available one can begin the visualization. If you don't have data then please note that MayaVi ships with a simple example of a heart CT scan data in the VTK data format (this should be in the `examples/` directory). The dataset used is a structured points dataset with a scalar field. More VTK, PLOT3D and EnSight data samples should be available from the VTK download page <`http://www.vtk.org/get-software.php`>.

The first thing to do is load an appropriate data file. Visit the **File** menu and choose the appropriate menu item for the data you have and select the file you want from the resulting GUI. Once you do this, you will notice that the control panel will show a *DataVizManager* at the topmost list and you will see a lot of controls below this. The control panel is described in some detail in this section. If you aren't familiar with that section yet, this might be a good time to review it.

Once the data file is opened a dialog box will pop up that allows one to configure the datafile. One can choose the appropriate data field one is interested in. This configuration window can be closed when unnecessary. To reconfigure the data use the **Configure Data** button provided in the control panel. Only one scalar attribute and one vector attribute is supported at a given time. Each *DataVizManager* that is seen represents a different file. One can open as many data files as one wants. Different data types can be loaded simultaneously too. Using a similar procedure one can import a simple VRML2 scene or even a 3D studio file. To close a VRML or 3D Studio file choose the appropriate file in the **Close** menu.

MayaVi also supports time series data. If the file name ends with an integer, MayaVi treats this integer as a time index. All files in the same directory as the loaded file are scanned. If any of them share the same pattern (without the last integer) as the name of the opened file, then the files are treated as part of the time series. These files are then sorted. The **Configure Data** GUI automatically addes a slider to switch between these time steps and an auto-sweep button to sweep through the time series.

Once the data file is read and a `DataVizManager` is created an instance of a `ModuleManager` is also created. At this point one has to load the `Filter`s and the `Module`s in order to do the visualization.

**7.3.4. Visualizing the Data.** Once the data file has been opened and the appropriate field attribute for both scalars and vectors has been chosen one can either filter the data or one can directly apply a module to the data and visualize it. MayaVi provides a large number of Modules and a few Filters. They are described in more detail subsequently.

In order to Filter the data one must use the **Visualize** menu and from the **Filters** sub-menu choose the appropriate filter. As soon as a filter is requested a popup window will appear that helps you configure the particular filter. Please note that it is not at all necessary to filter the data. If no filtering is required one does not need to load any filter. Even if a filter is used one can delete it at any time using the controls provided in the control panel.

In order to use a particular visualization module a procedure similar to the one for Filters is used. One merely uses the **Visualize** menu and from the **Modules** sub-menu chooses the appropriate Module. The module might take a little while to load. If there is some kind of error a warning dialog will attempt to describe the problem and hopefully the user can correct the situation.

7.3.4.1. *Navigating the Visualization.* It is important to be able to navigate the data and view it appropriately. In MayaVi this is achieved in one of two ways. Using the standard view mode buttons provided at the bottom of the visualization widget or by using the mouse to navigate through the visualization. The mouse based navigation is far more powerful and general purpose. The buttons however provide quick shortcuts to commonly desired views. The buttons and the visualization widget are shown in the illustration in an earlier chapter.

Navigating using the mouse. Mouse navigation is powerful but takes a little getting used to. It is relatively simple and with experience can be used easily. This section briefly describes how one can use the mouse to navigate through the data.

**Rotating the visualization:**

With 3D visualization it is important to be be able to rotate the visualized scene. In MayaVi this is achieved by first placing the mouse pointer on top of the visualization window. Then one keeps the left mouse button pressed and drags the mouse pointer in the direction one needs to rotate the scene. This is very much like rotating an actual object.

**Zooming in and out:**

To zoom in and out of the scene first one places the mouse pointer inside the visualization window. To zoom into the scene one keeps the the right mouse button pressed and drags the mouse upwards. To zoom out of the scene one keeps the right mouse button pressed and drags the mouse downwards.

**Panning the scene:**

To pan a scene implies translating the center of the rendered scene. In MayaVi this is done in two ways.

(1) By keeping the left mouse button pressed and simultaneously holding down the **Shift** key and dragging the mouse in the appropriate direction.
   (a) By keeping the middle mouse button pressed and dragging the mouse in the appropriate direction.

Just practice this a few times and you should get used to this pretty easily. This practically covers all that you need to know to be able to use MayaVi effectively. The best way to really learn about MayaVi is to explore the various options and try them out. Subsequent sections provide more details on the various menu's provided and the various modules and filters that are available.

**7.3.5. Picking data.** MayaVi supports data picking. While visualizing some data press the **p** or **P** key to pick the data point above the current mouse position. This will pop up a new window where the picked values will be displayed. The window also has a few controls that let you configure the type of picking you wish to perform. Please note that the picker will pick data on the actors that you have visualized on screen. It will attempt to find the nearest actor and pick the data on that. The picked point will be highlighted using an axes. The picker supports picking the following:

**Picking the nearest point:**

This option is the default and lets you pick the nearest point in the data. You could use this if you have data specified as point data. Note that when you pick using this option the picked location may not be exactly where you placed the mouse. The location will snap to the nearest available point. By changing the tolerance presented in the GUI you can control how near the point should be to the exact picked position.

**Picking the nearest cell:**

This option lets you pick the nearest cell in the data. You could use this if you have data specified as cell data. By changing the tolerance you can control how closely the picker finds the nearest cell.

**Picking an arbitrary point:**

This option lets you pick an arbitrary point in space that is not tied to the nearest cell or point. The motion of this picker is much smoother than the point or cell picker. However, the point that you pick will not be exactly at a point in the actual data. The results will therefore be interpolated using a probe filter.

**7.3.6. Configuring the lights.** MayaVi allows you to configure the lighting of the scene using a graphical utility. When the mouse is over the visualization frame of the window press the **l** or **L** key to open the light configuration kit. This will pop up a new window where you can configure as many as eight different lights. The default light is to have one light placed as a headlight. A headlight is a light that points directly ahead in the same direction as the camera. It is possible to change the position of the default light. It is also possible to turn on other lights, configure their elevation and azimuthal positions using the sliders provided, configure their intensity and color. To configure a particular light click on one of the conical glyphs that indicate a particular light. The GUI is fairly easy to follow. Experiment with it to become comfortable with it.

Note that if you save a visualization your light settings are also saved and when you reload the visualization these settings are restored.

**7.3.7. The Menus.** This section details the various menus that MayaVi provides. Almost all the menu items have hot keys associated with them. The underlined letter indicates the key sequence to be used. Consider the case of the **New Window** menu item (the letter $N$ is underlined) that is in the **File** menu (the letter $F$ is underlined). This can be reached by using the following key strokes. **Alt-F** followed by **N**. The Menu itself requires the **Alt** modifier but the menu item does not. The following are the various menus that MayaVi provides.

7.3.7.1. *File Menu.* The **File** menu provides the following menu items.

**New Window:**
This creates a new MayaVi visualization window that is completely independent of the first window. Any number of such windows can be created.

**Open:**
This provides a submenu containing two items.

**VTK file:**
This provides a GUI dialog from which a VTK file can be selected for opening. Once the file is opened a new `DataVizManager` is created and a GUI dialog for configuring the data file is provided.

**VTK XML file:**
This provides a GUI dialog from which a VTK XML file can be chosen. This is a new VTK format and is only available in VTK versions higher than 4.0.

**PLOT3D file:**
This provides a sub menu from which either a PLOT3D file containing single block or multi-block binary structured grid data can be selected for opening. Once the file is opened a new `DataVizManager` is created and a GUI dialog for configuring the data file is provided.

**EnSight case file:**
This provides a sub menu from which an EnSight case file can be opened.

**Import:**
This provides a submenu containing two items.

**VRML2 scene:**
This loads a VRML2 scene into the current visualization.

**3D Studio scene:**
This provides a menu containing all the VRML files that are already opened. The chosen VRML file is closed and the VRML files actors are removed from the rendered scene.

**Load:**
This provides a submenu containing three items.

**Visualization:**
This loads a saved *complete* visualization.

**ModuleManagers:**
This creates new `ModuleManager`s for the current `DataVizManager` and loads `ModuleManager`s that have been saved to a file into the newly created ones.

**ModuleManagers (Append):**
This is slightly different from the previous menu item and loads the first of the saved `ModuleManager`s into the current `ModuleManager` and for the subsequent saved `ModuleManager`s it creates new `ModuleManager`s and loads `ModuleManager`s from the saved file.

**Save:**
This provides a submenu containing four items.

**Entire Visualization:**
This saves the *entire* visualization configuration to a file such that it can be loaded by the **Load** menu's **Visualization** menu item.

**Current DataVizManager:**
This creates saves the current `DataVizManager` to a file. This can be loaded as a visualization.

**Current ModuleManager:**
This enables one to store the currently active `ModuleManager` into a file such that it can be loaded later.

**All ModuleManagers:**
This enables one to store the all the `ModuleManager`s for the currently active `DataVizManager` into a file such that it can be loaded later.

**Save Scene to:**
Provides a menu which in turn provides menu items to export the visualized scene to a Post Script file, PPM/BMP/TIFF/JPEG/PNG image, Open Inventor, Geomview OOGL, VRML and RenderMan RIB files. It is also possible to save the scene to a vector graphic via GL2PS <`http://www.geuz.org/gl2ps`>. This is only available if VTK is built with GL2PS support.

**Close:**
Provides submenu's to close the current `DataVizManager`. This means that all the `ModuleManager`s of that particular `DataVizManager` will also be deleted. It also provides menus to close the currently active VRML and 3D Studio scenes that have been imported.

**Close All:**
Close all the `DataVizManager`s and all the imported VRML2 and 3D Studio scenes.

**Exit:**
Close this particular MayaVi Window. If this is the only MayaVi window the application exits completely.

7.3.7.2. *Visualize Menu.* The **Visualize** menu provides the following menu items.

**Modules:**
Provides a sub-menu which contains a list of all available `Module`s. This list is dynamically generated based on the available modules.

**Filters:**
Provides a sub-menu which contains a list of all available `Filter`s. This list is dynamically generated based on the available filters.

**Pipeline browser:**
This creates a GUI that shows the entire VTK visualization pipeline. The objects in the pipeline can be configured by double clicking on the items. If there are a large number of objects this can become confusing to use and it would be better to use the pipeline segment browser configuration provided with the configuration for each `Module`

7.3.7.3. *Options Menu.* The **Options** menu provides the following menu items.

**Preferences:**
Provides a GUI using which one can edit the default preferences. The preferences allow one to set various default settings including foreground color, background color, default directory where the file related dialogs will open in initially. These options can be saved so that the next time MayaVi is started it will use these defaults. If you set the default directory to an empty one the directory that the file open/save dialogs will use will be intelligently chosen. The stereo rendering option enables stereo rendering in the MayaVi window. If the save current lighting option is set then the current lighting is saved as the default and used in all subsequent MayaVi visualizations.

The search path setting allows the user to specify a list of directories where user defined sources, modules and filters are made available. The search path is a ':'-separated string and is specified like the PYTHONPATH. '~', '~user' and '$VAR' are all expanded. Each of the directories specified in this string can have a `Sources/`, a `Modules/` and a `Filters/` directory inside where user defined sources, modules and filters can be stored. These modules and filters will be made available inside the **User** sub-menu of the **File/Open**, **Module** and **Filter** menus respectively. These sources, modules and filters can be used from the command line or from a Python interpreter session by using 'User.SourceName' (sources cannot be specified from the command line), 'User.ModuleName' or 'User.FilterName'. When creating user defined sources or modules or filters make sure that the name of the module is the same as the name of the class that defines the particular object.

**Configure RenderWindow:**
Provides a simple GUI to configure the Visualization RenderWindow. This is also where one can change the stereo rendering options.

**Change Foreground:**
Allows one to change the default foreground color.

**Change Background:**
Allows one to change the default background color.

**Show Debug window:**
Pops up a small window where debug messages are printed. This is very useful if you run into trouble and want to know what happened. It also shows the function call sequence. The messages are also printed to stderr.

**Show Control Panel:**

Toggles the visibility of the control panel. This can be very useful when you want to do a full screen visualization.

**Reload Modules:**

This function reloads all the currently loaded Python modules. This is *very* useful while debugging a new feature for some module that one is creating. One may also see funny behavior for already instantiated objects.

7.3.7.4. *Help Menu.* The **Help** menu provides the following menu items.

**About:**

Displays a few details about MayaVi.

**Users Guide:**

Opens a web browser and displays this MayaVi users guide.

**Home page:**

Opens a web browser and displays the MayaVi home page.

**7.3.8. Module Documentation.** The following are the list of provided Modules along with a brief description.

**Axes:**

This module creates and manages a set of three axes for your data. The class uses a vtkCubeAxesActor2D.

**BandedSurfaceMap:**

Displays a surface map with special contouring using the vtkBandedPolyDataContourFilter. This contour filter produces filled contours of the same color between two contour lines rather than either a continuous color distribution or just individual contour lines. It should work for any input dataset. It is best used for 2d surfaces. Note that one can either specify a total number of contours between the minimum and maximum values by entering a single integer or specify the individual contours by specifying a Python list/tuple in the GUI.

**ContourGridPlane:**

This module shows a grid plane of the given input structured, rectilinear or structured points grid with the scalar data either as a color map or as contour lines. This works only for structured grid, structured point and rectilinear grid data. Note that one can either specify a total number of contours between the minimum and maximum values by entering a single integer or specify the individual contours by specifying a Python list/tuple in the GUI.

**CustomGridPlane:**

This module shows a grid plane of the given input grid. The plane can be shown as a wireframe or coloured surface with or without scalar visibility and contour lines. The module basically wraps around the vtk*GeometryFilters. This module enables one to completely configure the grid plane. It works only for structured grid, structured point and rectilinear grid datasets. Note that one can either specify a total number of contours between the minimum and maximum values by entering a single integer or specify the individual contours by specifying a Python list/tuple in the GUI.

**Glyph:**

This module displays glyphs scaled and colored as per the input data. This will work for any dataset and can be used for both scalar and vector data.

**GridPlane:**

This module shows a grid plane of the given input grid. The plane can be shown as a wireframe or coloured surface with or without scalar visibility. This works only for structured grid, structured point and rectilinear grid data. Useful for debugging and displaying your created grid.

**HedgeHog:**

This module shows the given vector data as a 'hedge hog' plot. The lines can be colored based on the input scalar data. This class should work with any dataset.

**IsoSurface:**

This module shows an iso-surface of scalar data. This will work for any dataset.

**Labels:**

Displays text labels of input data. When instantiated, the class can be given a module name (the same name as listed in the Modules GUI) or an index of the module (starting from 0) in the current module manager. If this is not provided the module will ask the user to choose a particular module or choose filtered data. The module will then generate text lables for the data in the chosen module and display it. The module provides many configuration options. It also lets one turn on and off the use of a vtkSelectVisiblePoints filter. Using this filter will cause the module to only display visible points. Note that if the module that is being labeled has changed significantly or is deleted

this Labels module will have to be updated by changing one of the settings (like the RandomModeOn check button) to a different value and then back to the original one. Alternatively, choose the module to be labeled again.

**Locator:**
This module creates a 'Locator' axis, that can be used to mark a three dimensional point in your data.

**Outline:**
Displays an Outline for any data input.

**PolyData:**
Displays any input polydata, nothing fancy.

**ScalarCutPlane:**
This module plots scalar data on a cut plane either as a color map or with contour lines. This will work for any dataset. Note that one can either specify a total number of contours between the minimum and maximum values by entering a single integer or specify the individual contours by specifying a Python list/tuple in the GUI.

**Streamlines:**
This module makes it possible to view streamlines, streamtubes, and stream ribbons for any type of vector data. Any number of point sources can be added and deleted. A fairly powerful UI is provided. This module should work with any dataset.

**StructuredGridOutline:**
Displays an Outline for a structured grid.

**SurfaceMap:**
Displays a surface map of any data. It should work for any dataset but is best if used for 2d surfaces (polydata and unstructured surfaces). Note that one can either specify a total number of contours between the minimum and maximum values by entering a single integer or specify the individual contours by specifying a Python list/tuple in the GUI.

**TensorGlyphs:**
This module displays glyphs, scaled and colored as per the tensor data. This will work for any dataset.

**Text:**
Displays simple text on the screen. The text properties and position are configurable. The text can also be multi-line if newlines are embedded in it.

**VectorCutPlane:**
This module displays cone glyphs scaled and colored as per the vector or scalar data on cut plane. This will work for any dataset.

**VelocityVector:**
This module displays cone or arrow glyphs scaled and colored as per the vector data. This will work for any dataset.

**Volume:**
This Volume module allows one to view a structured points dataset with either unsigned char or short data as a volume. The module also provides a powerful GUI to edit the Color Transfer Function (CTF). You can drag the mouse with different buttons to change the colors. The following are the mouse buttons and key combinations that can be used to edit the CTF – red curve: Button-1, green curve: Button-2/Control-Button-1, blue curve: Button-3/Control-Button-2, alpha/opacity: Shift-Button-1. It is possible to use either the vtkVolumeRayCastMapper or the vtkVolumeTextureMapper2D. It is also possible to choose among various ray cast functions.

**WarpVectorCutPlane:**
This module takes a cut plane and warps it using a vtkWarpVector as per the vector times a scale factor. This will work for any dataset.

**7.3.9. Filter Documentation.** The following are the list of provided Filters along with a brief description.

**CellToPointData:**
This class produces PointData given an input that contains CellData. This is useful because many of VTK's algorithms work best with PointData. The filter basically wraps the vtkCellDataToPointData class.

**CutPlane:**
This filter takes a cut plane of any given input data set. It interpolates the attributes onto a plane. The position and orientation of the plane are configurable using a GUI.

**Delaunay2D:**
This filter wraps around the vtkDelaunay2D filter and lets you do 2D triangulation of a collection of points. The key parameters are Tolerance and the Alpha value. Tolerance gives the criteria for joining neighbouring data points and alpha is the threshold for the circumference of a caluculated triangulated polygon.

**Delaunay3D:**
This filter wraps around the vtkDelaunay3D filter and lets you do 3D triangulation of a collection of points. The key parameters are Tolerance and the Alpha value. Tolerance gives the criteria for joining neighbouring data points and alpha is the threshold for the circumference of a caluculated triangulated polygon.

**ExtractGrid:**
Wraps vtkExtractGrid (structured grid), vtkExtractVOI (imagedata/structured points) and vtkExtractRectilinearGrid (rectilinear grids). These filters enable one to select a portion of, or subsample an input dataset. Depending on the input data the appropriate filter is used.

**ExtractTensorComponents:**
This wraps the vtkExtractTensorComponents filter and allows one to select any of the nine components or the effective stress or the determinant from an input tensor data set. This will work for any dataset.

**ExtractUnstructuredGrid:**
This wraps the vtkExtractUnstructuredGrid filter. From the VTK docs: vtkExtractUnstructuredGrid is a general-purpose filter to extract geometry (and associated data) from an unstructured grid dataset. The extraction process is controlled by specifying a range of point ids, cell ids, or a bounding box (referred to as 'Extent'). Those cells lying within these regions are sent to the output. The user has the choice of merging coincident points (Merging is on) or using the original point set (Merging is off).

**ExtractVectorComponents:**
This wraps the vtkExtractVectorComponents filter and allows one to select any of the three components of an input vector data attribute.

**ExtractVectorNorm:**
This wraps the vtkVectorNorm filter and produces an output scalar data with the magnitude of the vector.

**MaskPoints:**
This wraps the vtkMaskPoints filter. The problem with this filter is that its output is Polygonal data. This means that if you add this filter to a ModuleManager with visualizations apart from HedgeHog or other velocity vector data you won't see anything! If that happens create another ModuleManager and show the other visualizations there. Also, this means that this filter should be typically inserted at the end of the list of filters.

**PolyDataNormals:**
This wraps the vtkPolyDataNormals filter. vtkPolyDataNormals is a filter that computes point normals for a polygonal mesh. This filter tries its best to massage the input data to a suitable form. Its output is a vtkPolyData object. Computing the normals is very useful when one wants a smoother looking surface.

**StructuredPointsProbe:**
A useful filter that can be used to probe any dataset using a Structured Points dataset. The filter also allows one to convert the scalar data to an unsigned short array so that the scalars can be used for volume visualization.

**Threshold:**
This wraps the vtkThreshold filter. The problem with this filter is that its output is an Unstructured Grid. This means that if you add this filter to a ModuleManager of a gridded dataset and you have a few grid planes, then your grid planes won't show anymore. If that happens create another ModuleManager and show the grid planes there acting on unfiltered data.

**UserDefined:**
This filter wraps around a user specified filter and lets one experiment with VTK filters that are not yet part of MayaVi. By default if the class is instantiated it will ask the user for the VTK class to wrap around. If passed a valid VTK class name it will try to use that particular class.

**WarpScalar:**
This wraps the vtkWarpScalar filter. vtkWarpScalar is a filter that modifies point coordinates by moving points along point normals by the scalar amount times the scale factor. Useful for creating carpet or x-y-z plots.

**WarpVector:**
Warps the geometry using the vtkWarpVector filter. vtkWarpVector is a filter that modifies point coordinates by moving points along vector times the scale factor. Useful for showing flow profiles or mechanical deformation.

## 7.4. Using MayaVi from Python

If you have installed MayaVi from the sources and are not using a binary release, then you can use MayaVi as a Python module. This chapter details how you can use MayaVi as a Python module. If you are looking for a powerful, interactive, cross-platform Python interpreter you might be interested in IPython. <http://ipython.scipy.org>

**7.4.1. An example.** Its very easy using MayaVi as a Python module. Thanks to Tkinter, it is also possible to use MayaVi from the Python interpreter. This means that one can script MayaVi! This is a pretty powerful and useful feature. To illustrate using MayaVi as a module and its scriptability, we will consider a few simple examples where the user generates some data and a VTK file and then uses MayaVi to visualize the data.

7.4.1.1. *Generating some data.*

```
>>> # generate the data.
>>> from Numeric import *
>>> import scipy
>>> x = (arange(50.0)-25)/2.0
>>> y = (arange(50.0)-25)/2.0
>>> r = sqrt(x[:,NewAxis]**2+y**2)
>>> z = 5.0*scipy.special.j0(r)  # Bessel function of order 0
>>> # now dump the data to a VTK file.
>>> import pyvtk
>>> # Flatten the 2D array data as per VTK's requirements.
>>> z1 = reshape(transpose(z), (-1,))
>>> point_data = pyvtk.PointData(pyvtk.Scalars(z1))
>>> grid = pyvtk.StructuredPoints((50,50, 1), (-12.5, -12.5, 0), (0.5, 0.5, 1))
>>> data = pyvtk.VtkData(grid, point_data)
>>> data.tofile('/tmp/test.vtk')
```

The above example uses the Numeric <http://numpy.sourceforge.net>, SciPy <http://www.scipy.org> and pyVtk <http://cens.ioc.ee/projects/pyvtk/> modules. Please note the step where `z1` is obtained from `z`. This step is done to correctly flatten the two dimensional array `z`. The problem with Numeric arrays and VTK data is that you have to be careful of the order of the data points. The way VTK reads data (for all the data formats that have a structure) is something like this:

```
>>> for k in range(n_z):
>>>     for j in range(n_y):
>>>         for i in range(n_x):
>>>             read_line()
```

This means that the x values must be iterated over first, the y values next and the z values last. If you simply flatten the 2D numeric array then this will not happen properly. By using `reshape(transpose(z), (-1,))` we ensure that the data points are specified in the correct order. The next step is to visualize the generated data.

7.4.1.2. *Visualize the generated data.*

```
>>> import mayavi
>>> v = mayavi.mayavi() # create a MayaVi window.
>>> d = v.open_vtk('/tmp/test.vtk', config=0) # open the data file.
>>> # The config option turns on/off showing a GUI control for the data/filter/module.
>>> # load the filters.
>>> f = v.load_filter('WarpScalar', config=0)
>>> n = v.load_filter('PolyDataNormals', 0)
>>> n.fil.SetFeatureAngle (45) # configure the normals.
```

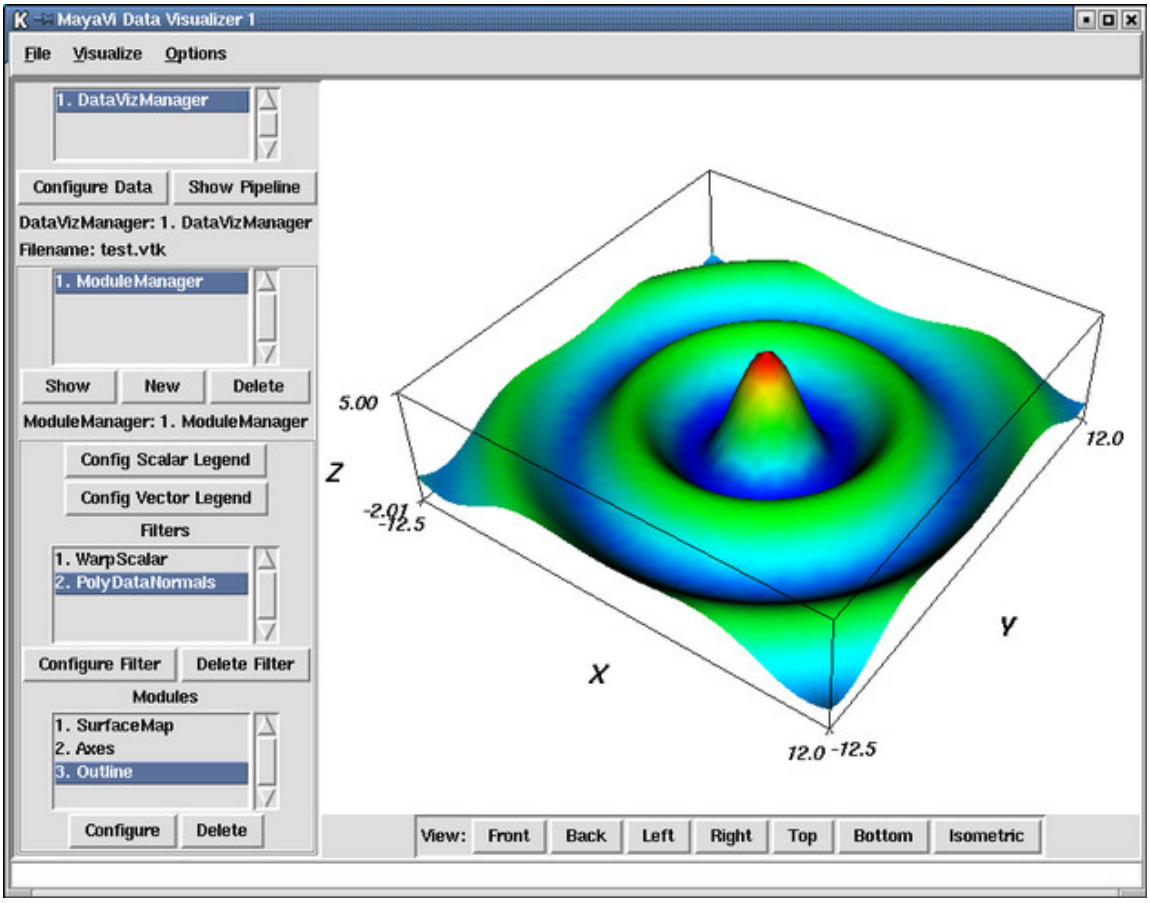



FIGURE 7.4.1. Surface plot

```
>>> # Load the necessary modules.
>>> m = v.load_module('SurfaceMap', 0)
>>> a = v.load_module('Axes', 0)
>>> a.axes.SetCornerOffset(0.0) # configure the axes module.
>>> o = v.load_module('Outline', 0)
>>> v.Render() # Re-render the scene.
```

The result of this is seen in the above figure. It is important to note that the Python interpreter will continue to remain interactive when MayaVi is running. In fact, it is possible to create an animation from the interpreter as done in the following.

```
>>> # now do some animation.
>>> import time
>>> for i in range (0, 10):
...  f.fil.SetScaleFactor(i*0.1)
...  v.Render()
...  v.renwin.save_png('/tmp/anim...')

...  time.sleep(1)
>>>
```

The above example saves the screen each iteration to a PNG image. One will need VTK 4.0 for PNG support. These images can be later used by some other utility to create a movie. It is therefore possible to create very useful visualizations from within the Python interpreter.

7.4.1.3. *Using VTK data objects.* There are times when the user has created a VTK data object that needs to be visualized. MayaVi has a special data handler for such cases. The following shows how this can be used. The example itself uses a VTK file but the data could have also been generated using other means.

```
>>> # import VTK
>>> import vtkpython
>>> # create some data.
>>> reader = vtkpython.vtkStructuredPointsReader()
>>> reader.SetFileName('/tmp/test.vtk')
>>> reader.Update()
>>> data = reader.GetOutput()  # this is a vtkStructuredPoints object.
>>> import mayavi
>>> v = mayavi.mayavi() # create a MayaVi window
>>> v.open_vtk_data(data) # load the data from the vtkStructuredPoints object.
>>> f = v.load_filter('WarpScalar', 0)
>>> # Load other filters and modules...
```

The above example uses a `vtkStructuredPoints` as the input. Other types can also be used as the input. The other valid types are: `vtkRectilinearGrid`, `vtkStructuredGrid`, `vtkUnstructuredGrid` and `vtkPolyData`. Any of these objects can be used as an input and then visualized using MayaVi.

Right now the best way to find out what functions are available etc. would be to read the sources or use pydoc to browse through the code. Experimenting with MayaVi from the interpreter is also a good idea and will be highly educative.

7.4.1.4. *Standalone MayaVi scripts.* After interactively exploring MayaVi from the interpreter one usually would like to run these in a non-interactive fashion. That is you'd like to create a Python script that invokes MayaVi. The easiest way to do it is as shown in the following simple example

```
import mayavi
v = mayavi.mayavi()
v.load_visualization('heart.mv')
# Do whatever you please with the MayaVi window.

# To make the MayaVi window interact with the user and wait
# till it is closed to proceed, do the following:
v.master.wait_window()

# Now once the previous window is closed if you need
# to open another do this:
v = mayavi.mayavi()
d = v.open_vtk('file.vtk')
# etc.
v.master.wait_window()

# Once the MayaVi window is closed the program will exit.
```

As can be seen above, it is easy to use code from an interactive session in a standalone Python script. It is also possible to script MayaVi in the following manner.

```
import Tkinter
r = Tkinter.Tk()
r.withdraw()
import mayavi
v = mayavi.Main.MayaViTkGUI(r)
v.load_visualization('heart.mv')
# Do whatever you please with the MayaVi window.

# now do this to start the Tk event loop.
```

```
root.mainloop()
# Once the MayaVi window is closed the program will exit.
```

This is an alternative way to do use MayaVi from Python scripts. This might be helpful if you have used `Tkinter` and know how to use it. However, the first approach is a lot easier.

**7.4.2. Useful Python Modules.** This section describes some other useful modules that are released as part of MayaVi but are not necessarily part of the core MayaVi module/application. The module `ivtk` is described in the next section. The MayaVi package also contains a sub-package called `tools`. This directory contains miscellaneous but useful tools that use or are related to MayaVi. This is described subsequently.

**7.4.3. The Interactive VTK module.** It is very nice to be able to use and experiment with VTK from the Python interpreter. In order to make this easier I've written a simple module that uses some of the MayaVi classes. This makes using VTK from Python very pleasant. The module is called `ivtk` which stands for interactive VTK. `ivtk` provides the following features.

- An easy to use VTK actor viewer that has menus to save the scene, change background, show a help browser, show a pipeline browser etc.
- A simple class documentation search tool/browser that lets you search for arbitrary strings in the VTK class documentation and lets you browse the VTK class documentation.
- An easy to use GUI to configure VTK objects using the `vtkPipeline.ConfigVtkObj` module.
- An integrated picker that can be activated by pressing the **p** or **P** keys. This picker functions the same way as the MayaVi picker.
- An integrated light configuration kit that can be activated by pressing the **l** or **L** keys. This light configuration functions the same way as the MayaVi light kit.

The help browser allows one to search for arbitrary strings in the VTK class documentation. 'and' and 'or' keywords are supported and this makes searching for specific things easier. If a search is successful a list of matching classes is returned. Clicking on a class will pop up a window with the particular class documentation. It is also possible to search for a particular class name. All classes matching the searched name will be shown. The searching is case insensitive.

Here is a sample session that illustrates how `ivtk` can be used. A simple cone example is shown.

```
>>> from mayavi import ivtk
>>> from vtkpython import *
>>> c = vtkConeSource()
>>> m = vtkPolyDataMapper()
>>> m.SetInput(c.GetOutput())
>>> a = vtkActor()
>>> a.SetMapper(m)
>>> v = ivtk.create_viewer() # or ivtk.viewer()
# this creates the easy to use render window that can be used from
# the interpreter.  It has several useful menus.

>>> v.AddActors(a)    # add actor(s) to viewer
>>> v.config(c)       # pops up a GUI configuration for object.
>>> v.doc(c)          # pops up class documentation for object.
>>> v.help_browser()  # pops up a help browser where you can search!
>>> v.RemoveActors(a) # remove actor(s) from viewer.
```

The `AddActors/RemoveActors` method can be passed a list/tuple or a single actor. All of the passed actors will be added/removed to the `vtkRenderWindow`. The `config` method provides an easy to use GUI to configure the passed VTK object. The viewer also provides menus to save the rendered scene and also provides a menu to open a VTK Pipeline browser that can be used to browse the VTK pipeline and configure objects in it.

Even without creating the actor viewer it is possible to use the help browser and the configure code as shown below.

```
>>> from mayavi import ivtk
>>> d = ivtk.doc_browser()
# pops up a standalone searcheable VTK class help browser.
```

```
>>> from vtkpython import *
>>> c = vtkConeSource()
>>> ivtk.doc(c)            # pops up class documentation for c
>>> ivtk.doc('vtkObject')  # class documentation for vtkObject.
>>> ivtk.config(c)         # configure object with GUI.
```

The module is fairly well documented and one should look at the module for more information. However, the above information should suffice if one wants to start using the module.

7.4.3.1. *The MayaVi `tools` sub-package.* MayaVi has a `tools` sub-package that contains useful modules that use or are related to MayaVi. The following modules are present currently.

The `imv` package. The `imv` module provides Matlab-like one liners that make it easy to visualize data from the Python interpreter. It currently provides three useful functions. These are partially described below. A simple example is also provided below that. The `imv` module is well documented so please read the documentation strings in the module for more details.

`surf(x, y, f):` This samples the function or 2D array $f$ along $x$ and $y$ and plots a 3D surface.

`view(arr):` Views 2D arrays as a structured points dataset. The view is set to the way we usually think of matrices with (0,0) at the top left of the screen.

`viewi(arr):` Views 2D arrays as a structured points dataset. The data is viewed as an image. This function is meant to be used with large arrays. For smaller arrays one should use the more powerful `view()` function. The implementation of this function is a bit of a hack and many of MayaVi's features cannot be used. For instance you cannot change the lookup table color and expect the color of the image to change.

`sampler(xa, ya, func):` Samples a function (`func`) at an array of ordered points (with equal spacing) and returns an array of scalars as per VTK's requirements for a structured points data set, i.e. x varying fastest and y varying next.

Here is a simple example of what can be done with the `imv` module.

```
>>> from Numeric import *
>>> from mayavi.tools import imv

>>> # surf example.
>>> def f(x, y):
...   return sin(x*y)/(x*y)
>>> x = arange(-5., 5.05, 0.05)
>>> y = arange(-5., 5.05, 0.05)
>>> v = imv.surf(x, y, f)

>>> # view/viewi example.
>>> z1 = fromfunction(lambda i,j:i+j, (128,256))
>>> v1 = imv.view(z1)

>>> z2 = fromfunction(lambda i,j:i+j, (512, 512))
>>> v2 = imv.viewi(z2)
```

## 7.5. Scripted examples

Load John's dataset, make a mayavi script out of his vtk one
volumetric rendering
time animation
slicing a 4d dataset

CHAPTER 8

# 3D visualization with VTK

The Visualization Toolkit is a library for creating, analyzing, and visualizing 3D data, and is a high level library that sits on top of a low-level library like OpenGL. Because 3D interaction and visualization is so computationally intensive, video cards come with special processors to do computations for 3D geometry at the hardware level, and low-level software libraries like OpenGL are used to communicate with the video card. However, low level libraries are just that, and do not support the higher level geometrical concepts that describe the problem at hand; eg OpenGL has no concept of a cube – you can create a cube by making six rectangular faces placed in the proper positions – but you can't say "draw me a cube" [**43**].

That is where high level libraries like VTK come in. VTK is an enormously powerful and complex visualization library written in C++ that can drive low level 3D libraries such as Mesa, a pure software OpenGL implementation, and OpenGL itself. It relies heavily on principles of object oriented design, and can be plugged into all widely used graphical user interfaces: Cocoa, Win32, Tkinter, WX, GTK, FLTK and more. Wrappers of the C++ code exist for Python, Java and TCL. VTK provides python users the ability to do cross-platform, hardware accelerated rendering from the comfort of the python interpreter, all with better quality than money can buy.

VTK is a complex library – with over 1000 classes and a deep inheritance scheme, it has an API rivaling Java's in complexity. The system was initially developed by Bill Schroeder and Will Lorenson, who designed the library after working in visualization and animation for 10 years. The initial design identified only 25 core classes, and four software professionals spent 10 months designing the library before touching a keyboard! These original 25 classes still exist in the library today [**38**].

## 8.1. Hello world in VTK

We'll start with a minimal example that creates and displays a cube. The example below creates a cube. The src for this example is `examples/vtk_hello.py` and the output is shon in Figure `8.1.1`.

Listing 8.1

```
#!/usr/local/bin/python
import os
import vtk

# Create a rectangular cube.  VTK has a number of "source" classes to
# create cubes, cones, cylinders, spheres, grids, images and even
# mandlebrot sets!
```

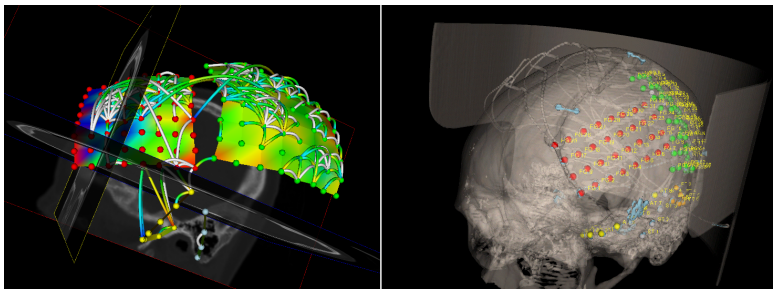


Figure 8.0.1. Screenshots from pbrain, a python application for localizing subdural electrodes and mapping statistical quantities from the ECoG onto their spatial coordinates which utilizes VTK for visualization.

```
cube = vtk.vtkCubeSource()
cube.SetXLength(10)
cube.SetYLength(5)
cube.SetZLength(20)
cube.SetCenter(1,2,3)

# Set up the mappers to extract data primitives.  The output of the
# mapper is polygon data that
mapper = vtk.vtkPolyDataMapper()
mapper.SetInput(cube.GetOutput())

# The actors are the objects that are added to the scene.  Here you
# can set properties of the actor, eg object (color, translucency,
# etc)
actor = vtk.vtkActor()
actor.SetMapper(mapper)
actor.GetProperty().SetColor(1,0,0)

# Rendering is the process of converting geometry (points, edges,
# polygon faces), lights, camera angles and so on to a 2D image view
# -- what you see on the screen.  The vtkRenderer is an abstract
# interface to concrete implementations, eg vtkOpenGLRenderer for
# hardware accelerated rendering
ren = vtk.vtkRenderer()
ren.AddActor(actor)

# The render window is a graphical user interface window in which the
# renderer above draws the 2D rendered image
renWin = vtk.vtkRenderWindow()
renWin.AddRenderer(ren)

# The render window interactor is a platform independent way for
# supporting GUI interaction -- mouse presses, keyborar events, mouse
# motion and so on
iren = vtk.vtkRenderWindowInteractor()
iren.SetRenderWindow(renWin)
renWin.SetSize(450,450)

# Ready, set, go!
iren.Initialize()
iren.Start()
```

EXERCISE 8.2. *Learning to fly.* It takes a little while to get used to the navigation controls in VTK. Every mouse button: left middle and right controls navigation, and there are multiple modes of interaction: camera versus actor and joystick versus trackball. For example, in camera mode, the left mouse button rotates the camera around its focal point, the middle mouse button pans the camera, and the right mouse button zooms. These controls have different meanings in actor mode (toggle camera and actor mode by pressing 'c' and toggle joystick versus trackball mode by pressing 'j'. Visit the `vtkInteractorStyle` page and read more about the ways you can interact with the data, and experiment with the differnent controls. Much like a video game, it takes a while before you are comfortable at the controls: we call this process *learning to fly.*[1]

EXERCISE 8.3. *Make a translucent sphere.* Modify `examples/vtk_hello.py` to add a translucent blue sphere to the to scene. *Translucency hint*: translucency is an actor property – as you see in the code above, you call `actor.GetProperty()` to get the actor property reference. Visit the VTK class docs web page and click on

[1] http://www.vtk.org/doc/nightly/html/classvtkInteractorStyle.html

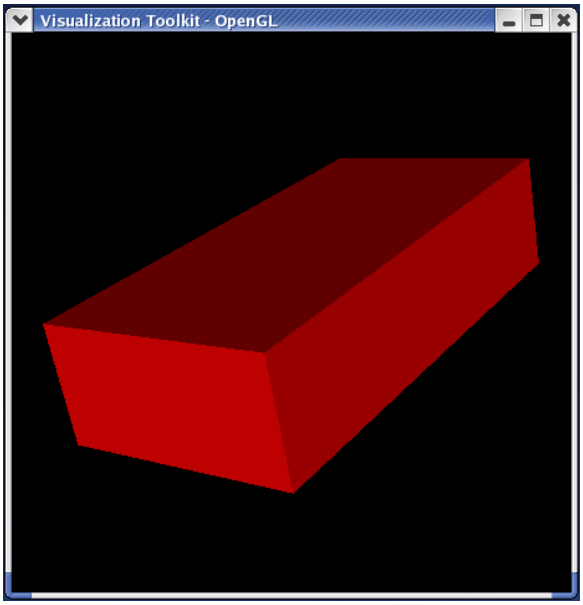



FIGURE 8.1.1. A Cube, brought to you by VTK

the vtk class `vtkActor` and find the method `GetProperty()`. If you click on the return value of this method (`vtkProperty`) you'll be taken to the right page. Read over the available methods to determine the appropriate one for setting the translucency and note all the other properties that you can control.

### 8.4. Working with medical image data

VTK has very strong support for medical image data, including volume data readers, DICOM readers, surface contour filters, plane slice widgets, and so on. In addition, the Insight Toolkit (ITK), which was developed by a consortium of universities and private companties, provides a large number state-of-the-art image segmentation and registration algorithms[**21**]. Like VTK, ITK is a large, sophisticated C++ library which comes with wrappers for a number of interpreted languages: Java, Tcl and Python. The first step in the pipeline to work with image/volume data is to create a reader, and VTK ships with readers and writers for all popular formats for medical image data, eg `vtkDICOMImageReader`, to read the ubiquitous DICOM format, and `vtkVolumeReader`, to work with raw binary images. Once the image is loaded, you can pass it to various filters or viewers. In the example below, we create an MRI viewer shown in Figure 8.4.1.

LISTING 8.2

```
#!/usr/local/bin/python
import os
import vtk
from WindowLevelInterface import WindowLevelInterface

# Create reader - you have total flexibility to specify the file
# naming pattern, the byte order, the size of the header and so on
reader = vtk.vtkVolume16Reader()
reader.SetDataDimensions(256,256)
reader.GetOutput().SetOrigin(0.0,0.0,0.0)
reader.SetFilePrefix('../data/images/r')
reader.SetFilePattern( '%s%d.ima')
reader.SetDataByteOrderToBigEndian()
reader.SetImageRange(1001,1060)
reader.SetDataSpacing(1.0,1.0,3.5)
reader.Update()
```

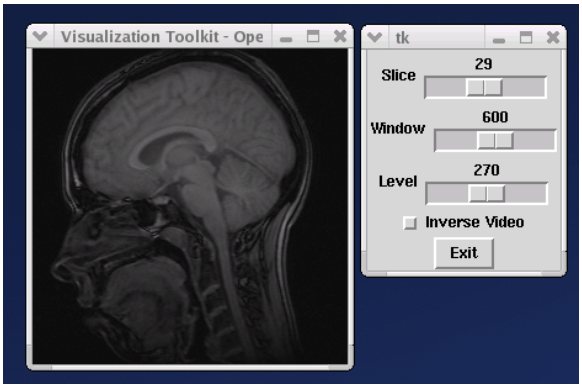



FIGURE 8.4.1. A simple slice viewer, in 30 lines of python

```
# VTK comes with a helper class to view slice data
viewer = vtk.vtkImageViewer()
viewer.SetInput(reader.GetOutput())
viewer.SetZSlice(30)
viewer.SetColorWindow(600)
viewer.SetColorLevel(270)
viewer.Render()
viewer.SetPosition(50,50)

# A helper class to set the window level, etc
WindowLevelInterface(viewer)
```

A common need in medical image data analysis and visualization is segmentation of different anatomical regions. VTK provides support for this with isosurface contouring via the patented marching cubes algorithm. Although marching cubes is patented, it is free for noncommerical use, though it is unclear whether use in an academic research setting qualifies as noncommercial.[2] However the patent is due to expire in the summer of 2005, so this issue will soon be moot. The marching cube algorithm takes a seed value which is an image intensity value, and generates one or more 2D level surfaces from the 3D volume data. For more sophisticated image segmentation routines, see the Insight Segmentation and Registration Toolkit, referenced above. In the example code below, we will read the image data set as we did in the slice viewer application above, and feed this data to the marching cubes algorithm. The output will be stored in a VTK data file, in this case a plain text file which stores vertex, edge and polygon information, and shown in Figure 8.4.2. These files can be read and the data in them manipulated, analyzed and visualized by external tools, eg MayaVi.

LISTING 8.3

```
#!/usr/local/bin/python
import os
import vtk
import colors


# Create the volume reader
reader = vtk.vtkVolume16Reader()
reader.SetDataDimensions(256,256)
reader.GetOutput().SetOrigin(0.0,0.0,0.0)
reader.SetFilePrefix('../data/images/r')
reader.SetFilePattern( '%s%d.ima')
reader.SetDataByteOrderToBigEndian()
reader.SetImageRange(1001,1060)
```

[2] http://www.imakenews.com/bakerbotts/e_article000166656.cfm

```
reader.SetDataSpacing(1.0,1.0,3.5)
reader.Update()

# Marching cubes generates iso-surfaces
iso = vtk.vtkMarchingCubes()
iso.SetInput(reader.GetOutput())

# We'll run it though the decimation filter to make it a little
# smaller.
decimate = vtk.vtkDecimate()
decimate.SetInput(iso.GetOutput())

# Some iso values
#   vessles : 120
#   cortex  : 100
#   face    :  20
iso.SetValue(0,100)
isoMapper = vtk.vtkPolyDataMapper()
isoMapper.SetInput(decimate.GetOutput())

# You can assign scalars to voxels, eg to map things onto the cortex
# Here we are just dislpaying the anatomy so we turn scalars off
isoMapper.ScalarVisibilityOff()

isoActor = vtk.vtkActor()
isoActor.SetMapper(isoMapper)
# this is the color of the surface
isoActor.GetProperty().SetColor(colors.antique_white)


# Let's save the data to a VTK file for external processing. The
# Update command is used to make sure the pipeline is up-to-date
# before writing

decimate.Update()
writer = vtk.vtkDataSetWriter()
writer.SetInput(decimate.GetOutput())
writer.SetFileName('../data/bighead.vtk')
writer.SetFileTypeToASCII()
writer.Write()

# Now visualize it; set up the renderer and add the actor
ren = vtk.vtkRenderer()
ren.AddActor(isoActor)
ren.SetBackground(0.2,0.3,0.4)

# Set up the render window and interactor
renWin = vtk.vtkRenderWindow()
renWin.AddRenderer(ren)
iren = vtk.vtkRenderWindowInteractor()
iren.SetRenderWindow(renWin)
renWin.SetSize(450,450)

# Ready, set, go!
```

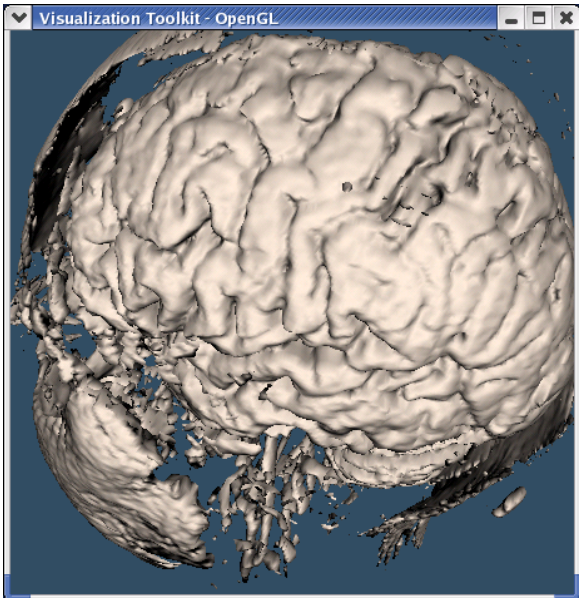



FIGURE 8.4.2. The cortical isosurface generated by a simple intensity based marching cubes application (40 lines of python listed above). More sophisticated image segmentation is available in the Insight Toolkit.

```
iren.Initialize()
iren.Start()
```

The possibilities for visualizing and analyzing 2D and 3D data with VTK and ITK are almost endless, and a person could spend many years mastering these libraries. Fortunately, there are many examples and several fine textbooks to speed you on your way. This introduction has only scratched the surface, using just a few of the 1000 or so classes available. Because the library is so large and daunting, and because the typical scientist doesn't have several years to master it, it is fortunate that MayaVi, a python application written by Prabhu Ramachandran, wraps this complexity into an easy to use graphical user interface. MayaVi is under active development, and will be a plugin component of the enthought envisage application framework for scientific computing in python.

CHAPTER 9

# Interfacing with external libraries

## 9.1. weave

Below is a listing of examples of weave use. This needs a lot of cleaning, as some of this code is very old and doesn't actually run with current weave.

```
#!/usr/bin/env python
"""Simple examples of weave use.

Code meant to be used for learning/testing, not production.

Fernando Perez <fperez@colorado.edu>
March 2002, updated 2003."""

from weave import inline,converters
from Numeric import *

#---------------------------------------------------------------------------
def simple_print(input):
    """Simple print test.

    Since there's a hard-coded printf %i in here, it will only work for
        ...numerical
    inputs (ints).  """

    # note in the printf that newlines must be passed as \\n:
    code = '''
    std::cout << "Printing from C++ (using std::cout) : "<<input<<std::endl;
    printf("And using C syntax (printf)    : %i\\n",input);
    '''
    inline(code,['input'],
           verbose=2)  # see inline docstring for details

def py_print(input):
    "Trivial printer, for timing."
    print "Input:",input

def c_print(input):
    "Trivial printer, for timing."
    code = """printf("Input: %i \\n",input);"""
    inline(code,['input'])

def cpp_print(input):
    "Trivial printer, for timing."
    code = """std::cout << "Input: " << input << std::endl;"""
    inline(code,['input'])
```

```
#-----------------------------------------------------------------------------
# Returning a scalar quantity computed from a Numeric array.
def trace(mat):
    """Return the trace of a matrix.
    """
    nrow,ncol = mat.shape
    code = \
"""
double tr=0.0;

for(int i=0;i<nrow;++i)
    tr += mat(i,i);
return_val = tr;
"""
    return inline(code,['mat','nrow','ncol'],
                  type_converters = converters.blitz)

#-----------------------------------------------------------------------------
# WRONG CODE: trace() version which modifies in-place a python scalar
# variable.  Note that this doesn't work, similarly to how in-place changes in
# python only work for mutable objects. Below is an example that does work.
def trace2(mat):
    """Return the trace of a matrix. WRONG CODE.
    """
    nrow,ncol = mat.shape
    tr = 0.0
    code = \
"""
for(int i=0;i<nrow;++i)
    tr += mat(i,i);
"""
    inline(code,['mat','nrow','ncol','tr'],
           type_converters = converters.blitz)
    return tr

#-----------------------------------------------------------------------------
# Operating in-place in an existing Numeric array. Contrary to trying to modify
# in-place a scalar, this works correctly.
def in_place_mult(num,mat):
    """In-place multiplication of a matrix by a scalar.
    """
    nrow,ncol = mat.shape
    code = \
"""
for(int i=0;i<nrow;++i)
    for(int j=0;j<ncol;++j)
        mat(i,j) *= num;

"""
    inline(code,['num','mat','nrow','ncol'],
           type_converters = converters.blitz)

#-----------------------------------------------------------------------------
# Pure Python version for checking.
```

```
def cross_product(a,b):
    """Cross product of two 3-d vectors.
    """
    cross = [0]*3
    cross[0] = a[1]*b[2]-a[2]*b[1]
    cross[1] = a[2]*b[0]-a[0]*b[2]
    cross[2] = a[0]*b[1]-a[1]*b[0]
    return array(cross)

#----------------------------------------------------------------------------
# Here we return a list from the C code. This is probably *much* slower than
# the python version, it's meant as an illustration and not as production
# code.
def cross_productC(a,b):
    """Cross product of two 3-d vectors.
    """
    # py::tuple or py::list both work equally well in this case.
    code = \
"""
py::tuple cross(3);

cross[0] = a(1)*b(2)-a(2)*b(1);
cross[1] = a(2)*b(0)-a(0)*b(2);
cross[2] = a(0)*b(1)-a(1)*b(0);
return_val = cross;
"""
    return array(inline(code,['a','b'],
                        type_converters = converters.blitz))

#----------------------------------------------------------------------------
# C version which accesses a pre-allocated NumPy vector.  Note: when using
# blitz, index access is done with (,,), not [][][]. In fact, [] indexing
# fails silently. See this and the next version for a comparison.
def cross_productC2(a,b):
    """Cross product of two 3-d vectors.
    """

    cross = zeros(3,a.typecode())
    code = \
"""
cross(0) = a(1)*b(2)-a(2)*b(1);
cross(1) = a(2)*b(0)-a(0)*b(2);
cross(2) = a(0)*b(1)-a(1)*b(0);
"""
    inline(code,['a','b','cross'],
           type_converters = converters.blitz)
    return cross

#----------------------------------------------------------------------------
# Just like the previous case, but now we don't use the blitz converters.
# Weave automagically does the type conversions for us.
def cross_productC3(a,b):
    """Cross product of two 3-d vectors.
    """
```

```
    cross = zeros(3,a.typecode())
    code = \
"""
cross[0] = a[1]*b[2]-a[2]*b[1];
cross[1] = a[2]*b[0]-a[0]*b[2];
cross[2] = a[0]*b[1]-a[1]*b[0];
"""
    inline(code,['a','b','cross'])

    return cross

#----------------------------------------------------------------------------
def dot_product(a,b):
    """Dot product of two vectors.

    Implemented in a funny (ridiculous) way to use support_code.

    I want to see if we can call another function from inside our own
    code. This would give us a crude way to implement better modularity by
    having global constants which include the raw code for whatever C
    functions we need to call in various places. These can then be included
    via support_code.

    The overhead is that the support code gets compiled in *every* dynamically
    generated module, but I'm not sure that's a big deal since the big
    compilation overhead seems to come from all the fancy C++ templating and
    whatnot.

    Later: ask Eric if there's a cleaner way to do this."""

    N = len(a)
    support = \
"""
double mult(double x,double y) {
    return x*y;
}
"""
    code = \
"""
double sum = 0.0;
for (int i=0;i<N;++i) {
    sum += mult(a(i),b(i));
}
return_val = sum;
"""
    return inline(code,['a','b','N'],
                  type_converters = converters.blitz,
                  support_code = support,
                  libraries = ['m'],
                  )

#----------------------------------------------------------------------------
def sumC(x):
```

```
    """Return the sum of the elements of a 1-d array.

    An example of how weave accesses a Numeric array without blitz.  """

    num_types = {Float:'double',
                 Float32:'float'}
    x_type = num_types[x.typecode()]

    code = """
           double result=0.0;
           double element;

           for (int i = 0; i < Nx[0]; i++){

               // Note the type of the pointer below is computed in python
               //element = *(%s *)(x->data+i*x->strides[0]);

               // Weave's magic does the above for us:
               element = x[i];

               result += element;
               std::cout << "Element " << i << " = " << element << "\\n";
           }
           std::cout << "size x " << Nx[0] << "\\n";

           return_val = result;
           """ % x_type;

    return inline(code,['x'],verbose=0)

#----------------------------------------------------------------------------
def Cglobals(arr):
    """How to pass data from function to function via globals.

    This allows the kind of 'over the head' parameter passing via globals
    which is ugly but necessary for using things like generic integrators in
    Numerical Recipes with aditional parameters.  """

    support = \
"""
// Declare globals here

/* These blitz guys must be accessed via pointers to avoid a costly copy.
Note that now the type is hardwired in. All python polymorphism is gone.  I
should look into whether this can be fixed by properly using blitz templating.
*/
blitz::Array<int, 1> *G_arr_pt;

// The global M will be visible in the "code" segment
int M = 99;

void aprint(int N) {
  std::cout << "In aprint()\\n";
  for (int i=0;i<N;++i)
```

```
    std::cout << "arr[" << i << "]=" << (*G_arr_pt)(i) << " ";
  std::cout << std::endl;
}

"""
    code = \
"""
// Get the passed array reference so the data becomes global
G_arr_pt = &arr;

std::cout << "global M=" << M << std::endl;
std::cout << "local N=" << N << std::endl;


std::cout << "First, print using the blitz internal printer:\\n";
std::cout << "all arr\\n";
std::cout << arr << std::endl;

std::cout << "all G_arr\\n";
std::cout << *G_arr_pt << std::endl;

std::cout << "now by loop\\n";

for (int i=0;i<N;++i)
  std::cout << "arr[" << i << "]=" << arr(i) << " ";
std::cout << std::endl;


std::cout << "Now calling aprint\\n";

aprint(N);
"""
    N = len(arr)
    return inline(code,['arr','N'],
                  type_converters = converters.blitz,
                  support_code = support,
                  libraries = ['m'],
                  verbose = 0,
                  )


#----------------------------------------------------------------------------
# Two trivial examples using the C math library follow.
def powC(x,n):
    """powC(x,n) -> x**n. Implemented using the C pow() function.
    """
    support = \
"""
#include <math.h>
"""
    code = \
"""
return_val = pow(x,n);
"""
```

```
    return inline(code,['x','n'],
                  type_converters = converters.blitz,
                  support_code = support,
                  libraries = ['m'],
                  )

# Some callback examples
def foo(x,y):
     print "In Python's foo:"
     print 'x',x
     print 'y',y
     return x

def cfoo(x,y):
      code = """
      printf("Attemtping to call back foo() from C...\\n");
      py::tuple foo_args(2);
      py::object z;  // This will hold the return value of foo()
      foo_args[0] = x;
      foo_args[1] = y;
      z = foo.call(foo_args);
      printf("Exiting C code.\\n");
      return_val = z;
      """
      return inline(code,"foo x y".split() )

x=99
y="Hello"

print "Pure python..."
z=foo(x,y)
print "foo returned:",z
print "\nVia weave..."
z=cfoo(x,y)
print "cfoo returned:",z

# Complex numbers
def complex_test():
    a = zeros((4,4),Complex)
    a[0,0] = 1+2j
    a[1,1] = 2+3.5j
    print 'Before\n',a
    code = \
"""
std::complex<double> i(0, 1);
std::cout << a(1,1) << std::endl;
a(2,2) = 3.0+4.5*i;
//a(2,2).imag = 4.5;
"""
    inline(code,['a'],type_converters = converters.blitz)
    print 'After\n',a

complex_test()
```

```
#--------------------------------------------------------------------------
def sinC(x):
    """sinC(x) -> sin(x). Implemented using the C sin() function.
    """
    support = \
"""
#include <math.h>
"""
    code = \
"""
return_val = sin(x);
"""
    return inline(code,['x'],
                  type_converters = converters.blitz,
                  support_code = support,
                  libraries = ['m'],
                  )

def in_place_multNum(num,mat):
    mat *= num


from weave import inline
class bunch: pass

def oaccess():
    x=bunch()

    x.a = 1

    code = """ // BROKEN!
    // Try to emulate Python's: print 'x.a',x.a
    std::cout << "x.a " << x.a << std::endl;
    """
    inline(code,['x'])

main2 = oaccess


def ttest():
    nrun = 10
    size = 6000
    mat = ones((size,size),'d')
    num = 5.6
    tNum = time_test(nrun,in_place_multNum,*(num,mat))
    print 'time Num',tNum
    tC = time_test(nrun,in_place_mult,*(num,mat))
    print 'time C',tC

def main():
    print 'Printing comparisons:'
    print '\nPassing an int - what the C was coded for:'
    simple_print(42)
```

```
print '\nNow passing a float. C++ is fine (cout<< takes care of things) but
    ... C fails:'
simple_print(42.1)
print '\nAnd a string. Again, C++ is ok and C fails:'
simple_print('Hello World!')

A = zeros((3,3),'d')

A[0,0],A[1,1],A[2,2] = 1,2.5,3.3

print '\nMatrix A:\n',A
print 'Trace by two methods. Second fails, see code for details.'
print '\ntr(A)=',trace(A)
print '\ntr(A)=',trace2(A)

a = 5.6
print '\nMultiplying A in place by %s:' % a
in_place_mult(a,A)
print A

# now some simple operations with 3-vectors.
a = array([4.3,1.5,5.6])
b = array([0.8,2.9,3.8])

print '\nPython and C versions follow. Results should be identical:'
print 'a =',a
print 'b =',b

print '\nsum(a_i) =',sum(a)
print 'sum(a_i) =',sumC(a)

print '\na.b =',dot(a,b)
print 'a.b =',dot_product(a,b)

print '\na x b =',cross_product(a,b)
print 'a x b =',cross_productC(a,b)

print '\nIn-place versions.'
print 'a x b =',cross_productC2(a,b)
print 'a x b =',cross_productC3(a,b)

print '\nSimple functions using the C math library:'
import math
x = 3.5
n = 4
theta = math.pi/4.
print '\nx**'+str(n)+'=',x**n
print 'x**'+str(n)+'=',powC(x,n)
print '\nsin('+str(theta)+')=',math.sin(theta)
print 'sin('+str(theta)+')=',sinC(theta)

print '\nGlobal variables and explicitly typed blitz arrays.'
x = array([4,5,6])
print 'x is a Numeric array:\nx=',x
```

```
    print 'Now using weave:'
    Cglobals (x)

if __name__ == '__main__':
    main()
```

## 9.2. swig

## 9.3. f2py

This is a rough set of notes on how to use f2py. It does NOT substitute the official manual, but is rather meant to be used alongside with it.

For any non-trivial poject involving f2py, one should also keep at hand Pierre Schnizer's excellent 'A short introduction to F2PY', available from http://fubphpc.tu-graz.ac.at/~pierre/f2py_tutorial.tar.gz

**9.3.1. Usage for the impatient.** Start by building a scratch signature file automatically from your Fortran sources (in this case all, you can choose only those .f files you need):

```
f2py -m MODULENAME -h MODULENAME.pyf *.f
```

This writes the file MODULENAME.pyf, making the best guesses it can from the Fortran sources. It builds an interface for the module to be accessed as 'import adap1d' from python.

You will then edit the .pyf file to fine-tune the python interface exhibited by the resulting extension. This means for example making unnecessary scratch areas or array dimensions hidden, or making certain parameters be optional and take a default value.

Then, write your setup.py file using distutils, and list the .pyf file along with the Fortran sources it is meant to wrap. f2py will build the module for you automatically, respecting all the interface specifications you made in the .pyf file.

This approach is ultimately far easier than trying to get all the declarations (especially dependencies) right through Cf2py directives in the Fortran sources. While that may seem appealing at first, experience seems to show that it's ultimately far more time-consuming and prone to subtle errors. Using this approach, the first f2py pass can do the bulk of the interface writing and only fine-tuning needs to be done manually. I would only recommend embedded Cf2py directives for very simple problems (where it works very well).

The only drawback of this approach is that the interface and the original Fortran source lie in different files, which need to be kept in sync. This increases a bit the chances of forgetting to update the .pyf file if the Fortran interface changes (adding a parameter, for example). However, the benefit of having explicit, clear control over f2py's behavior far outweighs this concern.

**9.3.2. Choosing a default compiler.** Set the FC_VENDOR environment variable. This will then prevent f2py from testing all the compilers it knows about.

**9.3.3. Using Cf2py directives.** For simpler cases you may choose to go the route of Cf2py directives. Below are some tips and examples for this approach.

Here's the signature of a simple Fortran routine:

```
      subroutine phipol(j,mm,nodes,wei,nn,x,phi,wrk)

      implicit real *8 (a-h, o-z)
      real *8 nodes(*),wei(*),x(*),wrk(*),phi(*)
      real *8 sum, one, two, half
```

The above is correctly handled by f2py, but it can't know what is meant to be input/output and what the relations between the various variables are (such as integers which are array dimensions). If we add the following f2py directives, the generated python interface is a lot nicer:

```
      subroutine phipol(j,mm,nodes,wei,nn,x,phi,wrk)
c
c       Lines with Cf2py in them are directives for f2py to generate a better
c       python interface.  These must come _before_ the Fortran variable
c       declarations so we can control the dimension of the arrays in Python.
c
c       Inputs:
Cf2py   integer check(0<=j && j<mm),depend(mm) :: j
```

```
Cf2py   real *8 dimension(mm),intent(in) :: nodes
Cf2py   real *8 dimension(mm),intent(in) :: wei
Cf2py   real *8 dimension(nn),intent(in) :: x
c
c       Outputs:
Cf2py   real *8 dimension(nn),intent(out),depend(nn) :: phi
c
c       Hidden args:
c       - scratch areas can be auto-generated by python
Cf2py   real *8 dimension(2*mm+2),intent(hide,cache),depend(mm) :: wrk
c       - array sizes can be auto-determined
Cf2py   integer intent(hide),depend(x):: nn=len(x)
Cf2py   integer intent(hide),depend(nodes) :: mm = len(nodes)
c
        implicit real *8 (a-h, o-z)
        real *8 nodes(*),wei(*),x(*),wrk(*),phi(*)
        real *8 sum, one, two, half
```

Some comments on the above:

- The f2py directives should come immediately after the 'subroutine' line and before the Fortran variable lines. This allows the f2py dimension directives to override the Fortran var(*) directives.
- If the Fortran code uses var(N) instead of var(*), the f2py directives can be placed after the Fortran declarations. This mode is preferred, as there is less redundancy overall. The result is much simpler:

```
        subroutine phipol(j,mm,nodes,wei,nn,x,phi,wrk)
c
c       Lines with Cf2py in them are directives for f2py to generate a better
c       python interface.  These must come _before_ the Fortran variable
c       declarations so we can control the dimension of the arrays in Python.
c
c       Inputs:
Cf2py   integer check(0<=j && j<mm),depend(mm) :: j
Cf2py   real *8 dimension(mm),intent(in) :: nodes
Cf2py   real *8 dimension(mm),intent(in) :: wei
Cf2py   real *8 dimension(nn),intent(in) :: x
c
c       Outputs:
Cf2py   real *8 dimension(nn),intent(out),depend(nn) :: phi
c
c       Hidden args:
c       - scratch areas can be auto-generated by python
Cf2py   real *8 dimension(2*mm+2),intent(hide,cache),depend(mm) :: wrk
c       - array sizes can be auto-determined
Cf2py   integer intent(hide),depend(x):: nn=len(x)
Cf2py   integer intent(hide),depend(nodes) :: mm = len(nodes)
c
        implicit real *8 (a-h, o-z)
        real *8 nodes(*),wei(*),x(*),wrk(*),phi(*)
        real *8 sum, one, two, half
```

Since python can automatically manage memory, it is possible to hide the need for manually passed 'work' areas. The C/python wrapper to the underlying fortran routine will allocate the memory for the needed work areas on the fly. This is done by specifying intent(hide,cache). 'hide' tells f2py to remove the variable from the argument list and 'cache' tells it to auto-generate it.

In cases where the allocation cost becomes a performance problem, one can remove the 'hide' part and make it an optional argument. In this case it will only be generated if not given. For this, the line above should be changed to:

```
Cf2py   real *8 dimension(2*mm+2), intent(cache), optional, depend(mm) :: wrk
```

Note that this should only be done after proving that the scratch areas are causing a performance problem. The `cache` directive causes f2py to keep cached copies of the scratch areas, so no unnecessary mallocs should be triggered.

Since f2py relies on Numeric arrays, all dimensions can be determined from the arrays themselves and it is not necessary to pass them explicitly.

With all this, the resulting f2py-generated docstring becomes:

```
phipol - Function signature:
  phi = phipol(j,nodes,wei,x)
Required arguments:
  j : input int
  nodes : input rank-1 array('d') with bounds (mm)
  wei : input rank-1 array('d') with bounds (mm)
  x : input rank-1 array('d') with bounds (nn)
Return objects:
  phi : rank-1 array('d') with bounds (nn)
```

**9.3.4. Debugging.** For debugging, use the –debug-capi option to f2py. This causes the extension modules to print detailed information while in operation. In distutils, this must be passed as an option in the f2py_options to the Extension constructor.

**9.3.5. Wrapping C codes with f2py.** Below is Pearu Peterson's (the f2py author) response to a question about using f2py to wrap existing C codes. While SWIG provides similar functionality and weave is perfect for inlining C, f2py seems to be an incredibly simple and convenient tool for wrapping C libraries.

Pearu's response follows:

For example, consider the following C file:

```
/* foo.c */
double foo(double *x, int n) {
  int i;
  double r = 0;
  for (i=0;i<n;++i)
    r += x[i];
  return r;
}
/* EOF foo.c */
```

To wrap the C function foo() with f2py, create the following signature file bar.pyf:

```
! -*- F90 -*-
python module bar
  interface
    real*8 function foo(x,n)
      intent(c) foo
      real*8 dimension(n),intent(in) :: x
      integer intent(c,hide),depend(x) :: n = len(x)
    end function foo
  end interface
end python module bar
! EOF bar.pyf
```

(see usersguide for more info about intent(c)) and run

```
f2py -c bar.pyf foo.c
```

Finally, in Python:

```
>>> import bar
>>> bar.foo([1,2,3])
6.0
```

**9.3.6. Passing offset arrays to Fortran routines.** It is possible to pass offset arrays (like pointers to the middle of other arrays) by using Numeric's slice notation.

The print_dvec function below simply prints its argument as "print*,'x',x". We show some examples of how it behaves with both 1 and 2-d arrays:

```
In [3]: x
Out[3]: array([ 2.8,  3.4,  4.1])
In [4]: tf.print_dvec(x)
 n 3
 x  2.8  3.4  4.1
In [5]: tf.print_dvec ?
Type:           fortran
String Form:    <fortran object at 0x8306fe8>
Namespace:      Currently not defined in user session.
Docstring:
    print_dvec - Function signature:
      print_dvec(x,[n])
    Required arguments:
      x : input rank-1 array('d') with bounds (n)
    Optional arguments:
      n := len(x) input int
In [6]: tf.print_dvec (x[1])
 n 1
 x  3.4
In [7]: tf.print_dvec (x[1:])
 n 2
 x  3.4  4.1
In [8]: A
Out[8]:
array([[ 3.5,  5.6,  8.2],
       [ 2.1,  4.5,  1.2],
       [ 6.3,  3.4,  3.1]])
In [9]: tf.print_dvec(A)
 n 9
 x  3.5  5.6  8.2  2.1  4.5  1.2  6.3  3.4  3.1
In [10]: A
Out[10]:
array([[ 3.5,  5.6,  8.2],
       [ 2.1,  4.5,  1.2],
       [ 6.3,  3.4,  3.1]])
In [11]: tf.print_dvec(A[1:])
 n 6
 x  2.1  4.5  1.2  6.3  3.4  3.1
In [12]: A[1:]
Out[12]:
array([[ 2.1,  4.5,  1.2],
       [ 6.3,  3.4,  3.1]])
In [13]: A[1:,1:]
Out[13]:
array([[ 4.5,  1.2],
       [ 3.4,  3.1]])
In [14]: tf.print_dvec(A[1:,1:])
 n 4
 x  4.5  1.2  3.4  3.1
```

**9.3.7. On matrix ordering and in-memory copies.** Numeric (which f2py relies on) is C-based, and therefore its arrays are stored in row-major order. Fortran stores its arrays in column-major order. This means that copying issues must be dealt with. Below we reproduce some comments from Pearu on this topic given in the f2py mailing list in June/2002:

> To avoid copying, you should create array that has internally Fortran data ordering. This is achived, for example, by reading/creating your data in Fortran ordering to Numeric array and then doing Numeric.transpose on that. Every f2py generated extension module provides also function
>
> has_column_major_storage

to check if an array is Fortran contiguous or not. If has_column_major_storage(arr) returns true then there will be no copying for the array arr if passed to f2py generated functions (assuming that the types are proper, of cource).

Also note that copying done by f2py generated interface is carried out in C on the raw data and therefore it is extremely fast compared to if you would make a copy in Python, even when using Numeric. Tests with say 1000x1000 matrices show that there is no noticable performance hit when copying is carried out, in fact, sometimes making a copy may speed up things a bit – I was quite surprised about that myself.

So, I think, you should worry about copying only if the sizes of matrices are really large, say, larger than 5000x5000 and efficient memory usage is relevant. The time spent for copying is negligible even for large arrays provided that your computer has plenty of memory (>=256MB).

**9.3.8. Distutils.** Below is an example setup.py file which generates a Python extension module from Fortran90 sources and a .pyf interface file generated by f2py and later fine tuned.

```
#!/usr/bin/env python
"""Setup script for F2PY-processed, Fortran based extension modules.

A typical call is:

% ./setup.py install --home=~/usr

This will build and install the generated modules in ~/usr/lib/python.

If called with no args, the script defaults to the above call form (it
automatically adds the 'install --home=~/usr' options)."""

# Global variables for this extension:
name         = "mwadap_tools"  # name of the generated python extension (.so)
description  = "F2PY-wrapped MultiWavelet Tree Toolbox"
author       = "Fast Algorithms Group - CU Boulder"
author_email = "fperez@colorado.edu"

# Necessary sources, _including_ the .pyf interface file
sources = """
binary_decomp.f90 binexpandx.f90 bitsequence.f90 constructwv.f90
display_matrix.f90 findkeypos.f90 findlevel.f90 findnodx.f90 gauleg.f90
gauleg2.f90 gauleg3.f90 ihpsort.f90 invert_f2cmatrix.f90 keysequence2d.f90
level_of_nsi.f90 matmult.f90 plegnv.f90 plegvec.f90 r2norm.f90 xykeys.f90

mwadap_tools.pyf""".split()

# Additional libraries required by our extension module (these will be linked
# in with -l):
libraries = ['m']

# Set to true (1) to turn on Fortran/C API debugging (very verbose)
debug_capi = 0

#**************************************************************************
# Do not modify the code below unless you know what you are doing.

# Required modules
import sys,os
from os.path import expanduser,expandvars
```

```
from scipy_distutils.core import setup,Extension

expand_sh = lambda path: expanduser(expandvars(path))

# Additional directories for libraries (besides the compiler's defaults)
fc_vendor = os.environ.get('FC_VENDOR','Gnu').lower()
library_dirs = ["~/usr/lib/"+fc_vendor]

# Modify default arguments (if none are supplied) to install in ~/usr
if len(sys.argv)==1:
    default_args = 'install --home=~/usr'
    print '*** Adding default arguments to setup:',default_args
    sys.argv += default_args.split()  # it must be a list

# Additional options specific to f2py:
f2py_options = []
if debug_capi:
    f2py_options.append('--debug-capi')

# Define the extension module(s)
extension = Extension(name = name,
                      sources = sources,
                      libraries = libraries,
                      library_dirs = map(expand_sh,library_dirs),
                      f2py_options = f2py_options,
                      )

# Call the actual building/installation routine, in usual distutils form.
setup(name = name,
      description  = description,
      author       = author,
      author_email = author_email,
      ext_modules  = [extension],
      )
```

### 9.4. Others

boost, pyrex, cxx

### 9.5. Distributing standalone applications

py2exe, mcmillan installer

# Bibliography


[1] *Python Success Stories: 8 True Tales of Flexibility, Speed, and Improved Productivity*. O'Reilly Associates, 2002.

[2] *Python Success Stories Volume II: 12 More True Tales*. O'Reilly Associates, 2005.

[3] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999.

[4] K. Arnold, J. G., and D. Holmes. *Java(TM) Programming Language*. Addison-Wesley Professional, 2005.

[5] J. W. Backus et al. Algol 60 — revised report on the algorithmic language. *Communications of the ACM*, 6(1):1–17, January 1963.

[6] C. H. Barker and W. P. Healy. Statistical analysis of oil spill response options: A NOAA-U.S. Navy joint project. In *Proceedings of the International Oil Spill Conference*, pages 883–890, 2001.

[7] P. Barrett, J.D. Hunter, and P. Greenfield. Matplotlib - A portable Python plotting package. In *Astronomical Data Analysis Software & Systems XIV.*, 2004.

[8] D. Beasley. *Python Essential Reference*. New Riders Publishing, 2nd edition, 2001.

[9] David Beazley. SWIG and automated C/C++ scripting extensions. *Dr. Dobb's Journal of Software Tools*, 23(2):30, 32, 34–36, 100, February 1998.

[10] Thomas J. Bergin, Richard G. Gibson, and Richard G. Gibson. *History of Programming Languages*. Addison-Wesley Professional, 1996.

[11] J.B. Buckheit and D.L. Donoho. *Wavelets and Statistics*, chapter WaveLab and Reproducible Research. Springer-Verlag, 1995.

[12] A. Butterfield, V. Vedagiri, E. Lang, C. Lawrence, M. J. Wakefield, A. Isaev, and G. A. Huttley. Pyevolve: A toolkit for statistical modelling of molecular evolution. *BMC Bioinformatics*, 5(1):1, 2004.

[13] P. F. Dubois, K. Hinsen, and J. Hugunin. Numerical Python. *Computers in Physics*, 10(3):262–267, May/June 1996.

[14] Paul F. Dubois and T.-Y. Yang. Scientific programming: Extending Python. *Computers in Physics*, 10(4):359–??, ???? 1996.

[15] Jr. Drake G. Van Rossum, F. L., editor. *An Introduction to Python*. Network Theory Ltd., 2003.

[16] A. Goldberg and D. Robson. *Smalltalk 80 : The Language*. Addison-Wesley Professional, 1989.

[17] Duane C. Hanselman and Bruce L. Littlefield. *Mastering MATLAB 7*. Prentice Hall, 2004.

[18] Jim Hugunin. Python and java - The best of both worlds. In *In Proceedings of the 6th International Python Conference*, pages 11–20, 1997.

[19] JD Hunter, J Reimer, DM Hanan, KE Hecox, and VL Towle. Locating chronically implanted subdural electrodes using 3-D rendering. *Clinical Neurophysiology*, 2005.

[20] Gavin A Huttley. Modeling the impact of DNA methylation on the evolution of BRCA1 in mammals. *Mol Biol Evol*, 21(9):1760–8, 2004.

[21] L. Ibáñez and W. Schroeder. *The ITK Software Guide: The Insight Segmentation and Registration Toolkit*. Kitware, Inc., 2003.

[22] I. K. Kominis, T. W. Kornack, J.C.. Allred, and M. V. Romalis. A subfemtotesla multichannel atomic magnetometer. *Nature*, 422(6932):596–599, April 2003.

[23] T. W. Kornack and M. V. Romalis. Dynamics of two overlapping spin ensembles interacting by spin exchange. *Phys Rev Lett*, 89(25):253002, December 2002.

[24] A. Martelli. *Python in a Nutshell*. O'Reilly, 1st edition, 2003.

[25] A. Martelli and D. Ascher. *Python Cookbook*. O'Reilly, 1st edition, 2004.

[26] Peter Naur al. Revised report on the algorithmic language ALGOL 60. 6(1):1–17, January 1963.

[27] NOAA. *TAP II 1.2 User Manual*. National Oceanic and Atmospheric Administration, Hazardous Material Response Division, Seattle, WA, 2000.

[28] John K. Ousterhout. Scripting: Higher level programming for the 21st century. *IEEE Computer*, 1998.

[29] H. Parker-Hall and C. H. Barker. Do trajectories belong in area plans? a new approach in california using the trajectory analysis planner (TAP II). In *Proceedings of the International Oil Spill Conference*, pages 685–691, 2001.

[30] Fernando Pérez. IPython – an enhanced interactive Python shell, 2001. `http://ipython.scipy.org`.

[31] M. Pilgrim. *Dive into Python*. Apress, 1st edition, 2004.

[32] S. M. Ransom, J. W. T. Hessels, I. H. Stairs, P. C. C. Freire, F. Camilo, V. M. Kaspi, and D. L. Kaplan. Twenty-One Millisecond Pulsars in Terzan 5 Using the Green Bank Telescope. *Science*, 307:892–896, February 2005.

[33] S. M. Ransom, V. M. Kaspi, R. Ramachandran, P. Demorest, D. C. Backer, E. D. Pfahl, F. D. Ghigo, and D. L. Kaplan. Green Bank Telescope Measurement of the Systemic Velocity of the Double Pulsar Binary J0737-3039 and Implications for Its Formation. *Astrophysical Journal*, 609:L71–L74, July 2004.

[34] S. Rosen. *Programming Systems and Languages*, chapter Programming Systems and Languages–A Historical Survey. McGraw-Hill, New York, 1967.

[35] G. Van Rossum. *Python Library Reference*. To Excel Inc, 2001.

[36] M.F. Sanner. A component-based software environment for visualizing large macromolecular assemblies. *Structure*, 13:447–462, 2005.

[37] M.F. Sanner. *Encyclopedia of Genomics, Proteomics and Bioinformatics*, chapter Using the Python Programming Language for Bioinformatics. John Wiley & Sons, Ltd, 2005.

[38] W. Schoeder, K. Martin, and B Lorense. *The Visualization Toolkit: An Object Oriented Approach to 3D Graphics*. Kitware, Inc., 3rd edition, 2002.

[39] G. Stein. Python at Google. In *Pycon2005*, 2005.

[40] M Strous. Python - executable pseudocode. *PC Update*, 2001.

[41] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley Professional, 3rd edition edition, 2000.

[42] Bjarne Stroustrup. *The Design and Evolution of C++*. Addison-Wesley Professional, 1994.

[43] M. Woo, J. Neider, T. Davis, and D. Shreiner. *OpenGL Programming Guide: The Official Guide to Learning OpenGL*. Addison Wesley, 3rd edition, 1999.